Technology
Arts Sciences
TH Köln

An empirical evaluation of using the Swift language as the underlying technology of RESTful APIs

Bachelor's Thesis

written by

Teodora-Roxana Petrisor

Submitted at

TH Köln – University of Applied Sciences
Campus Gummersbach
Faculty of Computer Science and
Engineering Science

under the Study Program

Allgemeine Informatik
Matriculation number: 11093932

Examiner:	Prof. Dr. Heide Faeskorn-Woyke Technische Hochschule Köln
Second examiner:	Prof. Dr. Matthias Böhmer Technische Hochschule Köln

Gummersbach, July 11th 2016

Abstract

Teodora-Roxana Petrisor, Allgemeine Informatik, TH Köln – University of Applied Sciences
Abstract of Bachelor's Thesis, submitted July 11th 2016
An empirical evaluation of using the Swift language as the underlying technology of RESTful APIs

The purpose of the current thesis is to determine the appropriateness of using the Swift language as the underlying technology for the development of RESTful APIs in a Linux environment. The current paper describes the process of designing, implementing and testing individual RESTful API components based on Node.js, PHP, Python and Swift and seeks to determine whether Swift is a viable alternative.
The thesis begins by defining a methodology for implementing and testing individual RESTful API components based on Node.js, PHP, Python and Swift. It then proceeds to detail the implementation and testing processes, following with an analytic discussion regarding the advantages and drawbacks of using the Swift language as the underlying technology for RESTful APIs and server-side Linux-based applications in general.
Following the implementation of five individual API server components based on the aforementioned technologies and using the Docker container software, the resulting applications were tested using the Apache JMeter software and compared based on functionality, performance, efficiency, reliability, disk usage and ease of implementation. While the Swift-based implementations managed to meet the minimal set of requirements defined in the aforementioned methodology, they did not reach the same performance, efficiency nor reliability as their Node.js counterpart.
Based on the implementation process and on the results of the previously mentioned evaluation phase, it can be stated that the Swift language is not yet ready to be used in a production environment. However, its rapid evolution and potential for surpassing its competitors in the foreseeable future make it an ideal candidate for implementing RESTful APIs to be used in development environments.

Contents

1 Introduction

1.1 Motivation

In June 2014, Apple Inc. announced the release of a new programming language named Swift[1], which could replace the previously used Objective-C language when programming for the iOS, OS X and the newly introduced tvOS and watchOS platforms. Swift was described as being not only faster than its predecessor, but also more manageable due to its introduction of modern programming language concepts such as dot syntax and functional programming patterns. Over the course of the next year, Swift registered an increase in popularity of 50,5%, while Objective-C displayed a drop of approximately 85,8%[2].
Apple's announcement in December 2015 that Swift 2.2 would be released as open-source[3] only advanced the promotion of the language within additional developer circles by breaking its dependency on any proprietary operating systems and making it available for a multitude of further purposes. Swift could now be used not only to write applications for Apple's platforms, such as iOS and OS X, but also to develop software programs running in Linux and Windows environments.
One such possible use case consists in developing a server-side application capable of running in a Linux-based environment and of providing a RESTful[4] API[5] to authorized clients. Due to the frequency with which this use case is encountered in the current software ecosystem, with most major software companies such as Facebook and Google providing at least one application interface in order to enable the development of third-party applications with their data, the current thesis will focus on the development of RESTful APIs as a representative for the more general development of server-side applications with the aforementioned technologies.
As Swift packages are already available for Ubuntu systems, the possibility of implementing such an interface is not subject to question. However, the decision to use Swift as the underlying technology for such an API, as opposed to other more tested

[1] Apple's announcement video for the release of the Swift language can be found under *Introduction to Swift*. Apple Inc. June 2, 2014. URL: `https://developer.apple.com/videos/wwdc2014/` (visited on 03/16/2016)
[2] *TIOBE Index for Swift*. Tiobe. Mar. 16, 2016. URL: `http://www.tiobe.com/tiobe_index?page=Swift` (visited on 03/16/2016); *TIOBE Index for Objective-C*. Tiobe. Mar. 16, 2016. URL: `http://www.tiobe.com/tiobe_index?page=Objective-C` (visited on 03/16/2016)
[3] *Apple Releases Swift as Open Source*. Apple Inc. Dec. 3, 2015. URL: `http://www.apple.com/pr/library/2015/12/03Apple-Releases-Swift-as-Open-Source.html` (visited on 03/16/2016)
[4] Representational State Transfer (REST)
[5] Application Program Interface (API)

and more widely used technologies, remains open for debate. It is therefore necessary to analyze how Swift-based APIs measure up against interfaces implemented with more popular technologies, such as Node.js, PHP or Python.

1.2 Preface

The following thesis discusses the appropriateness of using Swift as the underlying technology of a RESTful API, as opposed to Node.js, PHP or Python. It firstly consists of an overview of the aforementioned technologies and their use in the implementation of server-side applications. Furthermore, it provides a methodology for implementing a viable testing environment for these technologies, as well as a description of how this was achieved in the practical component pertaining to this thesis. Based on the results gathered in a subsequent testing phase, the thesis provides an analytic discussion of the appropriateness of using Swift as a server-side language as opposed to the aforementioned alternative technologies.
Chapter 2 provides an outline of the concept and architecture of a RESTful API and discusses how it enables the communication between multiple clients and one server application. It also consists of summaries of each of the previously mentioned technologies and their particularities and potential benefits when being used in a server-side context.
Chapter 3 consists in defining a methodology for constructing and testing API implementations having the previously mentioned four technologies as their foundation. Furthermore, it defines the criteria upon which any subsequent evaluation of the resulting API implementations should be based.
According to the aforementioned methodology, Chapter 4 describes the implementation process of the testing environment and of the individual APIs, detailing which tools and frameworks were used in order to meet the previously specified requirements.
The subsequent chapter revolves around the testing phase of the four technologies regarding their appropriateness to be used for the implementation of server-side applications. It defines the test cases to be used on each of the API implementations, as well as describes their execution. Chapter 5 also provides the results of the tests for each of the implementations, summarized based on test case, underlying technology and additional influencing parameters.
Chapter 6 provides an analytic discussion of the appropriateness of using Swift as a server-side technology as opposed to Node.js, PHP or Python, taking into consideration both the particularities of the language itself, and the results depicted in the previous chapter. The discussion aims to clarify the current position of Swift among other technologies in the context of server-side applications, as well as its potential for future improvement and use outside of Apple's ecosystem.
Lastly, Chapter 7 seeks to provide a general statement regarding the appropriateness

of choosing Swift as a server-side technology when implementing RESTful APIs.
This paper addresses various aspects of API implementation and server-side application development using multiple programming languages and paradigms. A comprehensive understanding thereof assumes a basic knowledge of Unix processes, client-server applications and object-oriented, as well as functional programming languages.

2 Fundamentals

The following chapter provides an outline of the architecture of client-server communication based on RESTful APIs, followed by an overview of each of the four server-side technologies considered for this thesis, namely Node.js, PHP, Python and Swift. These technologies will be presented both from the perspective of their underlying languages' syntax and library particularities, as well as from the viewpoint of their required dependencies in order to be used for the implementation of server-side applications.

2.1 API-based client-server communication

In many cases of software development projects, the logic of an application can be divided in two components, namely a central and a peripheral one, having as an effect the encapsulation of responsibilities, as well as the distribution of the workload across multiple entities. Separating these two components on a software and even hardware level is therefore a commonly encountered software architecture structure referred to as the client-server model[6]. By conforming to the characteristics of this model, a software application will divide its logic in two sections, namely a centralized, server component responsible for providing access to shared resources or services, and a peripheral, client component responsible for requesting these resources and services and for interpreting them in such a way that is relevant for the end user of the application. A concrete example of a software system based on this model is a mobile application with access to external data. While the application running on the mobile device itself represents the client side of the system, of which there can be multiple instances, the database system running on an external host represents the server side.
In order for a software system based on the client-server model to correctly function, the communication between both sides must be supported. The client and server elements of an application are more commonly implemented on separate hardware components. Thus, their communication usually takes place over a network, as depicted in Figure 2.1:

[6] A detailed description of the client-server model as defined by the Oracle Corporation can be found under *Client/Server Architecture*. Oracle Corporation. June 16, 2016. URL: `https://docs.oracle.com/cd/A57673_01/DOC/server/doc/SCN73/ch20.htm` (visited on 06/16/2016)

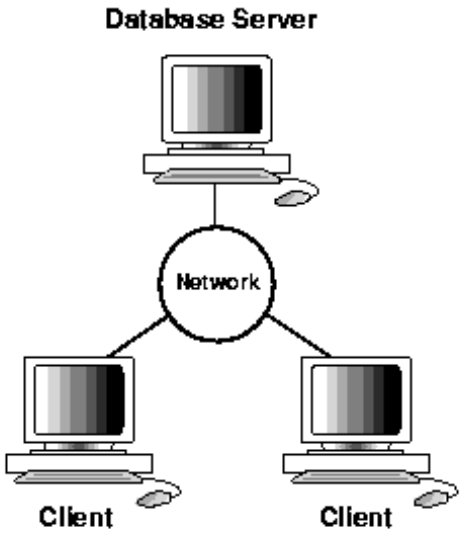


Figure 2.1: Application conforming to the client-server model distributed on separate machines and enabling communication over a network[7].

The communication between the two main components primarily consists of requests sent by the client component to the server, which is subsequently responsible for providing a correct response. The client does not need to be concerned with the server's implementation of the logic necessary to provide a response, but rather can maintain the perspective of the server component as representing a black box[8]. In order for a successful exchange of data or services between the two components to occur, the client and server need only implement the same protocol when communicating with each other. This protocol is responsible for specifying the content and the formatting of the client's request and of the server's response, thus providing a formal set of rules to govern the communication between the two components. Examples of such a protocol are FTP[9] or SMTP[10]. In order to provide further constraints to the inter-component communication, the server may provide an API, ensuring a strict specification of the content exchanged between itself and the client component.
For the purpose of this thesis, the current chapter will restrict itself to describing the concept of an API constructed over the HTTP[11] protocol and conforming to the REST software architectural style. REST defines a set of architectural properties, such as performance, scalability, portability and reliability, which can be realized by an API by conforming to the following formal constraints[12]:

[7] Graphics source: https://docs.oracle.com/cd/B10501_01/server.920/a96524/c07dstpr.htm (visited on 30/06/2016)
[8] In the context of computer science, a system described as a black box can be seen in terms of its input and output, without the need of any additional knowledge regarding its internal logic
[9] File Transfer Protocol (FTP)
[10] Simple Mail Transfer Protocol (SMTP)
[11] Hypertext Transfer Protocol (HTTP)
[12] See Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* University of California, Irvine, 2000, Chapter 5

1. Client-server
   The architecture of a RESTful software system must conform to the client-server model described in the previous paragraphs.

2. Stateless
   The underlying communication protocol of a RESTful software system must be stateless, i.e. each new request must be treated as independent and unrelated to any other requests. One such protocol is the Hypertext Transfer Protocol, or HTTP, which is used in data communication over the World-Wide-Web[13].

3. Cacheable
   Server responses must define themselves as cacheable or not in order to maximize performance and reduce redundant client-server communication.

4. Layered system
   A RESTful solution can be comprised of multiple architectural layers unaware of any other layers beyond their immediate neighbors within the layer hierarchy. For example, a client layer should not be able to distinguish between its connectivity status to the end server component and to an intermediary component responsible for forwarding information. This ensures that a RESTful service can be easily scaled if the need arises.

5. Uniform interface
   Server-side implementations must be decoupled from the services they provide, thus allowing for the client and server components to evolve independently from one another. The communication between the client and the server is restricted to the defined uniform interface, which must conform to the four following constraints:

   a) Identification of resources
      Resources should be uniquely identified in client requests, for example by using URIs[14] in web-based APIs. The resource representations received by the client in the response are conceptually different from the actual resources of the server component. For example, resources residing in a database on the server component can be sent to the client in a JSON or XML format, regardless of their server-side data representation.

   b) Manipulation of resources through representations
      A representation of a resource, along with any provided metadata, must include enough information so that the client may modify or delete the resource itself.

[13] The request for comments defining the HTTP protocol can be found under *Hypertext Transfer Protocol – HTTP/1.1*. Network Working Group. June 1, 1999. URL: `https://tools.ietf.org/html/rfc2616` (visited on 06/16/2016)

[14] Universal Resource Identifier (URI)

c) Self-descriptive messages
All messages between the client and the server must include enough information in order to describe how the message should be processed.

d) Hypermedia as the engine of application state (HATEOAS)
Apart from statically defined entry points to the server application, a client component should interact with the server exclusively through actions dynamically identified by the server itself through hypermedia. That is, the client cannot assume the existence of any particular action for any particular resource, such as its potential for modification, without said particular action being statically defined by the API itself or being provided in a previous response by the server component. This ensures that a client component needs no prior knowledge apart from an understanding of hypermedia in order to successfully communicate with a server component.

The present thesis will describe and evaluate the implementation and performance of RESTful APIs constructed over the HTTP protocol and implemented using the four aforementioned technologies. This section has defined the requirements needed to be met by any API in order to conform to the REST architectural style and will be used as a basis for the implementation of the RESTful APIs using Node.js, PHP, Python and Swift. The following sections will provide an overview of the aforementioned technologies, both from the perspective of their syntax particularities and regarding their intended use in implementing server-side applications.

2.2 Node.js

Node.js[15] is a cross-platform[16] framework used for the implementation of server-side applications. Based upon the V8 JavaScript engine from Google[17], it has an asynchronous, event-driven architecture. At its core, a Node.js application consists of an event loop responsible for processing incoming calls. The process runs on a single thread and uses non-blocking I/O calls[18] together with callback functions.
Executing parallel calls is handled through a thread pool. Whenever a new call is made to the main event loop thread, it is forwarded to a task queue, from which it is subsequently pulled and executed whenever a new thread becomes available. Upon

[15] A full documentation can be found under: *Node.js v5.11.1 Documentation*. Node.js Foundation. June 16, 2016. URL: `https://nodejs.org/dist/latest-v5.x/docs/api/` (visited on 06/16/2016)

[16] Node.js offers support for the major operating systems Linux, OS X and Microsoft Windows, as well as for several other Linux-based systems. For a list of all available download options, see: https://nodejs.org/dist/v4.4.2/ (visited on 30/06/2016)

[17] An overview of the V8 engine can be found under: https://developers.google.com/v8/ (visited on 30/06/2016)

[18] Non-blocking or asynchronous Input/Output operations are a type of I/O processing which allows the execution of other processing tasks before an input or output operation is completed.

the completion of the task, its corresponding callback function is called on the main thread of the Node.js application.
Due to the non-blocking nature of incoming calls to the server-side application, Node.js allows the development of scalable network applications without the concern for deadlocks[19]. Furthermore, despite its use of a single thread, the thread pool operating in the background and the use of third-party libraries such as pm2[20] in order to create clusters both allow for vertical scaling of the application by increasing the number of the server component's CPUs[21]. Furthermore, many popular Node.js modules such as Express.js[22] or Socket.IO[23] are written in JavaScript, enabling web developers who are familiar with the language to easily switch between programming the client-side and the server-side component of their application.
A notable particularity of the JavaScript language on which Node.js is based is its designation of functions as first-class citizens. That is, functions can be passed as parameters and returned as function results just as easily as any other variable types, as opposed to other languages such as Java. On the other hand, JavaScript does not by default support the concept of classes, but rather provides the possibility of simulating object-oriented programming with the help of prototypes and functions. This leads to the JavaScript language being more appropriate for the development of event-driven applications and not for the implementation of object-oriented architectures.

2.3 PHP

PHP[24] is a scripting language mainly used for web development in conjunction with HTML code to deliver dynamic websites to the requesting client[25]. The code itself is usually processed by a PHP interpreter, which is generally implemented as a module on the server component of the software system. Despite its main use to embed dynamic content in web applications, PHP can also be used to implement a server-side component responsible for reacting to inputs from the client and delivering an appropriately formatted response, for example in JSON.
As opposed to JavaScript, PHP provides full support for classes and objects, enabling developers versed in object-oriented programming to construct complex software ar-

[19] A common problem encountered in multiprocessing systems where two programs wait for each other to finish and thus never complete their tasks.
[20] The pm2 tool is documented under *PM2 - Advanced Node.js process manager*. Keymetrics. June 30, 2016. URL: `http://pm2.keymetrics.io` (visited on 06/30/2016)
[21] Central Processing Unit (CPU)
[22] The documentation for the Express framework can be found under *Express - Node.js web application framework*. Node.js Foundation. June 30, 2016. URL: `http://expressjs.com` (visited on 06/30/2016)
[23] The documentation for the Socket.IO framework can be found under *Socket.IO*. June 30, 2016. URL: `http://socket.io` (visited on 06/30/2016)
[24] PHP: Hypertext Preprocessor (PHP)
[25] The full documentation for PHP can be found under *PHP Manual*. The PHP Group. June 15, 2016. URL: `http://php.net/manual/en/` (visited on 06/16/2016)

chitectures. Its interpreter is also widely included in most web servers, removing the need for installing additional technologies in order to implement server-side applications.

2.4 Python

Python is a dynamic and interpreted programming language[26] used for the implementation of a wide variety of applications, including server-side applications. It supports multiple programming paradigms, such as object-oriented and structured programming, at the same time providing support for functional and aspect-oriented programming. Python interpreters are available for all major operating systems, including Linux, allowing for its use in implementing the logic of the server-side component of a RESTful system.
As opposed to the two previously mentioned languages, Python is a minimalist programming language that aims for readability and clean code. It also provides full support for object-oriented, functional and imperative programming, as well as being able to interact with external modules written in other languages, such as C or C++. Python is generally used as the main programming language in large projects featuring a high number of developers with differing backgrounds, in order to facilitate communication and cooperation.

2.5 Swift

Swift is a newly open-source language introduced by Apple Inc.[27], capable of supporting multiple programming paradigms, such as protocol-oriented, object-oriented and functional programming. Unlike the previous three languages, Swift is a compiled programming language which must be translated into machine code by the LLVM compiler before the application's logic can be executed.
Upon its release as an open source language, Swift became available for download not only for the OS X platform, but also for Ubuntu 14.04 and 15.10. This has allowed Swift to be used not only for the development of iOS and OS X applications as its predecessor[28], but also to be used as the foundation of server-side applications, such as the implementation of the server component of a RESTful software system. The syntax of Swift differs dramatically from that of its predecessor, replacing the use of square brackets in order to call object methods with the more familiar dot syntax encountered in most modern programming languages, such as Java or C#.

[26] The full documentation for the Python language can be found under *Python 3.5.1 documentation*. Python Software Foundation. June 10, 2016. URL: `https://docs.python.org/3/` (visited on 06/16/2016)

[27] An overview of the Swift language can be found under https://swift.org/about/ (visited on 30/06/2016)

[28] Objective-C is seen as Swift's predecessor, being the standard language for developing iOS and OS X applications before Swift was released

Moreover, as in the case of JavaScript, Swift functions are considered first-class citizens and can be passed around like any other variable types. The syntax of Swift is designed to support fast programming and easily readable code, as well as to make full use of functional programming concepts such as closures. At the same time, Swift also provides full support for object-oriented and protocol-oriented programming.

3 Methodology

The purpose of the current thesis is to analyze the appropriateness of using Swift in developing server-side applications, particularly in implementing RESTful APIs in comparison with Node.js, PHP and Python. In order to provide a correct assessment of each technology used in development, a methodology for the implementation and testing of the APIs must be specified and followed. The following chapter will specify a set of requirements for each of the implemented APIs and define a set of criteria based on which their subsequent analysis will be performed. Furthermore, the methodology will describe the process of setting up a viable testing environment for the aforementioned API implementations, a strategy for testing the individual resulting applications and an approach for the final analysis of their underlying technologies.

3.1 API definition

As the focus of this study is the appropriateness of using Swift for the development of RESTful APIs in comparison with Node.js, PHP and Python, the first necessary step in the analysis consists in using each of the four technologies to implement APIs conforming to the formal REST constraints specified in Chapter 2. Therefore, software systems based on Swift, Node.js, PHP and Python should be implemented based on the client-server architecture model, relying on a stateless communication protocol such as HTTP, having cacheable responses and a layered architecture, as well as providing a uniform interface through which data can be accessed and manipulated by the client component.
In order to ensure that an analysis of the resulting RESTful APIs will provide useful insight into the appropriateness of their underlying technologies, the applications should minimize any other differences other than their implementation technologies. The next step in the analysis therefore consists in defining a strict structure to be upheld by all implementations of the RESTful APIs. The definition of the structure should include all API endpoints available to the client component of the software architecture, as well as the protocol and calls through which they can be accessed. The communication between the client and server components of the implementations should also be identical in terms of response content and formatting.
Furthermore, as the analysis of the technologies pertains exclusively to the server component of the API implementations, all resulting software systems should share

the same client component. Hence, any interference in API performance related to the client side will be identical across all implementations, thus being rendered irrelevant in the final analysis.
Lastly, the client component of the application should not only be conceptually separated from the server component, but also reside on a different machine. Despite separating the two components on a software level with the use of virtual machines or container software, any resource-intensive client process would have to share its hardware resources with the server processes running on the same machine, leading to the possibility that a particularly stressful test would yield unreliable results. On the other hand, the server-side implementations of the API should reside on the same machine and have the same amount of resources at their disposal.
Based on the previously defined constraints, the following minimal set of requirements for the API implementations is defined:

- Each API should conform to the REST architecture style
- The server component of each API implementation should share the same underlying operating system and resources
- The communication between the client and server component should be carried out over the HTTP protocol
- Each API should expose one service call made available to the client through a GET request to the default `/` pathway.
- The exposed service call should return the same response for each implementation, encoded in JSON format, having the following contents:

```
1 {"response":"success","message":"successful message"}
```

- All implementations should share the same client component
- The client and server components should reside on separate machines

An analysis of the API implementations can only be undertaken following a confirmed adherence to the previously defined characteristics. It is therefore necessary to justify that each API implementation meets the specified requirements before the testing phase can be initiated.

3.2 Analysis points

In order to analyze and compare the resulting RESTful APIs implemented with the four aforementioned technologies, a set of criteria must be previously established

upon which any subsequent analysis and conclusions will be based. Figure 3.1 depicts a hierarchy of needs that each API should meet in order to be considered successful:

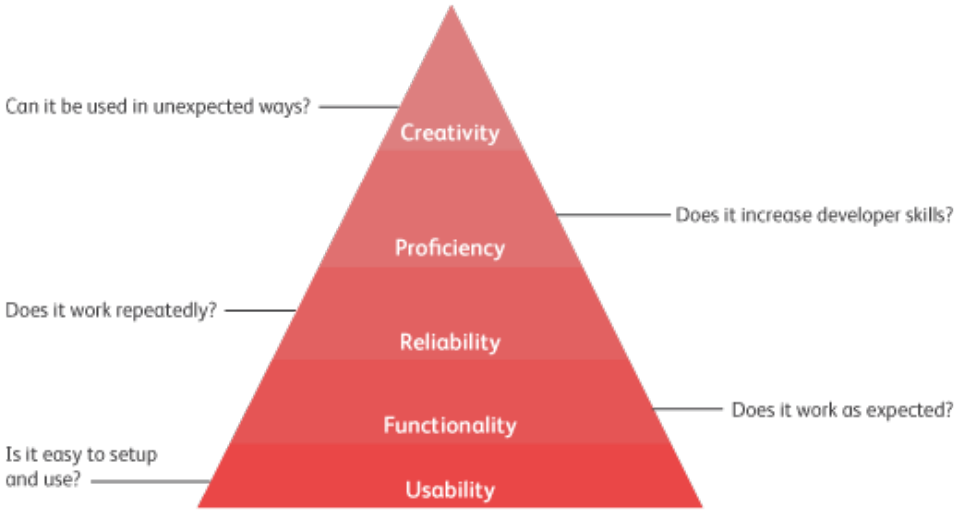


Figure 3.1: The requirements which should be fulfilled by an API in order to be considered successful[29].

The graphic depicts five requirements that an API should meet, in terms of their importance from the perspective of its user, namely usability, functionality, reliability, proficiency and creativity. When comparing external APIs, all five points should generally be taken into consideration in order to provide a correct conclusion regarding the superiority of one.
However, in the case of comparing APIs exclusively from the perspective of their underlying technologies, usability, proficiency and creativity become irrelevant points to consider. Whether or not an API is easy to setup and use by a developer is influenced by the structure of the API, as well as by the ease of the client component to access it. Both these characteristics have been required in the previous section to be identical across all implementations of the APIs and should therefore not be considered in the analysis process. Likewise, both proficiency and creativity are characteristics influenced by either the API structure, by the means through which the API calls can be accessed, or by the data received by the client component as a response. As these aspects have all been defined in the previous section of the current methodology as being invariable, their consideration in the analysis of the APIs also becomes unnecessary.
Therefore, the resulting API implementations will be evaluated from the perspective of the two remaining criteria, namely functionality and reliability. In terms of functionality, it will be sought to determine whether the implementations react as

[29] Graphics source: https://www.soapui.org/testing-dojo/world-of-api-testing/what-makes-api-testing-special.html (visited on 30/06/2016)

expected to the predefined calls to services, namely whether each service call returns the expected response content in the correct format to the client component of the software system. The implementations will also be evaluated in terms of reliability. The API with the lowest failure rate will be considered the most successful.
A further point not specified in the previous graphic will also figure in the analysis of the implementations, namely the performance of the API. Since all four APIs will be identical from the perspective of the server component and its communication with the client, any difference in response time can be directly attributed to their underlying technology and should factor into the final analysis.
Furthermore, the ease of implementation of each of the APIs, the size of the resulting server components, as well as the scalability of the resulting system will factor as additional consideration points in the final analysis.
The requirements defined in the previous section of the current chapter will not represent analysis points for the API implementations, but rather elimination criteria. Any server component failing to conform to the previously defined minimal set of requirements will be excluded from the final analysis of the current thesis and its underlying technology will be deemed inappropriate for developing RESTful APIs.

3.3 Environment setup

In order to correctly analyze each of the resulting APIs, an appropriate implementation and testing environment must be strictly defined and constructed for each of their underlying technologies. The following section describes how such an environment should be set up and what minimal requirements it should meet in order to provide accurate results for the testing phase of the following analysis.
As specified in section 3.1 of the current chapter, each of the implementations should consist of a server component running on one machine, and a shared client component implemented on a separate machine. In order to realize this separation not only on a software but also on a hardware level, two separate machines should each be connected through Ethernet cables to a network switch component responsible for enabling their communication. The use of a wired connection between the two machines as opposed to connecting them wirelessly to the same network is justified by the need to minimize any potentially perturbing factors, such as wireless signal interference or variation in signal strength, when performing the analysis of the API implementations.
Section 3.2 of the current chapter requires that the API implementations based on the four aforementioned technologies be analyzed not only based on their performance and failure rate, but also from the perspective of their potential scalability. It is therefore mandatory that the server components be tested by allocating a varying number of resources, such as base memory and number of CPUs. In order to meet this requirement, the current methodology recommends the use of virtualiza-

tion software for setting up the environment of the server components. Furthermore, all server components should be isolated from any external influences such as processes competing for the same resources. This can be achieved by shutting down all processes, including the other server components, on the machine when performing the testing phase, as well as by encapsulating each server-side component with either container software or virtual machines. Due to the low overhead of the former[30], as well as the fact that all implementations of the APIs should be based on the same operating system, as stated in section 3.1, the use of container software is recommended.
Furthermore, each API implementation should strictly consist of the software components necessary to meet the previously defined API requirements. Thus, each API should be encased within a separate container software based upon the same operating system and containing a single application implemented with one of the four technologies. Any unnecessary tools, programs or files should not be included in the container software, in order to prevent any undesirable impact on the analysis of the size and performance of the final implementations.
All implementations should also strictly conform to the previously defined functional requirements, namely that they should expose only one service call through an HTTP GET request, which should return an appropriate response encoded in JSON format. The implementation of any further functionality should be avoided, in order to limit any variation in performance or size due to differing program logic.
The client component of the API should be set up on a different machine and should be able to communicate over the HTTP protocol with all server components simultaneously. Preceding the execution of the formal testing phase of the analysis, each of the implementations should be informally tested for correct execution and response delivery. The subsequent test phase will only be carried out after all implementations have passed the previously mentioned informal test. Whether the implementations execute correctly and deliver the correct response should be determined by the party responsible for carrying out the subsequent test phase and analysis.

3.4 Evaluation

Each API implementation will be successively tested using the same tool and following the same test patterns. The testing tool should communicate with the server component of each implementation over the HTTP protocol and call the exposed service multiple times in a predetermined succession. Additionally, the tool should also provide support to simulate calls from multiple clients simultaneously, in order to determine how the individual implementations react to a pool of concurrent re-

[30] A comparison between container software and virtual machines can be found under Sudhi Seshachala. *Docker vs VMs*. Nov. 24, 2014. URL: `http://devops.com/2014/11/24/docker-vs-vms/` (visited on 06/16/2016)

quests. After ending a testing phase, the tool used for testing should provide the corresponding performance statistics, which will represent the raw data used in the subsequent analysis of the API implementations.
Each API should be tested using identical configurations, i.e. the same number of clients should make an identical number of calls to the different server components running on a system with the same hardware configuration. Each test should be carried out at least five times, with the numerical average of the performance statistics representing the base of the subsequent analysis. This decision was based on the fact that potential perturbing factors cannot be completely eliminated from the execution of one particular test case. In order to ensure that the results reflected the behavior of the server processes as much as possible, it was decided to carry out each test multiple times and to take the numerical average of their resulting performance as the base for any subsequent analysis. The choice of executing each test at least five times was also based on the total amount of time allocated to the practical part of the current thesis. A higher number of repetitions would lead to more reliable results, filtering out any statistical outliers. However, due to the high number of tests and the limited time allocated for their execution, it was decided that five repetitions would be sufficient in order to provide reliable statistical data. Furthermore, at least three types of identical test phases should be carried out, varying either in number of concurrent clients, number of performed calls, allocated memory or number of CPUs. This variation in test configuration was chosen in order to analyze how the underlying technologies performed when being allocated with a varying amount hardware resources while being confronted with an increasing number of concurrent requests.

3.5 Analysis

Following the execution of the test phase, the resulting performance statistics should be collected, documented and subsequently illustrated in an appropriate graphical representation for each test and implementation. The juxtaposition of the raw data according to each implementation technology will constitute the basis for the objective comparison of the APIs based on their performance, reliability and potential for scalability. The server components of the APIs should also be compared from the perspective of the remaining analysis points specified in section 3.2 of this chapter.
The collected data coupled with observations regarding the implementation process of the server components will serve as a foundation for an analytic discussion regarding the appropriateness of each of the four technologies for the implementation of RESTful APIs.

4 Implementation

The following chapter will present the systematic process of implementing the aforementioned APIs based on Node.js, PHP, Python and Swift respectively according to the methodology defined in Chapter 3.

4.1 Network configuration

According to the requirements defined in the previous chapter, two machines were used for the implementation of the testing environment. The machine used for housing the server processes was a MacBook Pro, having one 2.4 GHz Intel Core i7 processor with four cores, two 4 GB 1333 MHz DDR3 memory modules and running the OS X operating system, version 10.11.3. The second machine, used for running the client component of the API, was an HP ProBook 450 G0, having a 2.2 GHz Intel Core i7 processor with two cores, an 8 GB 2201 MHz memory module and running the Microsoft Windows 7 Professional operating system (64-Bit), version 6.1.7601 Service Pack 1 Build 7601.
In order to connect the two machines to a single switch network component, as per the requirements specified in the previous chapter, two Category 5E cables[31] were used. The switch component used as a middle point to establish communication between the two machines was the internal gigabit Ethernet switch of a FRITZ!Box 3390 router[32]. The choice of a router as opposed to a simple switch component was justified by the availability of the device, as well as by its support for the same functionality as a switch device. As this router not only allows the connection between two computer networks, but also the connection of multiple devices within a single network, its functionality extends that of a switch device. Having a total of four Ethernet ports, the FRITZ!Box 3390 device could therefore be used in conjunction with the Ethernet cables to connect the two aforementioned devices and act as a communication bridge between them. Once connected to the router, each machine received an IP address through which they could be addressed within the defined network.

[31] described in the ISO/IEC 11801 international standard
[32] The specifications of the device can be found under:
http://en.avm.de/products/fritzbox/fritzbox-3390/technical-data/ (visited on 30/06/2016)

4.2 Docker as a container software

Using software systems in order to embed and run applications in an isolated and controlled environment is not a recent idea, with the concept and use of virtual machines dating back to the 1960s[33]. Though useful, virtual machines present a significant overhead in terms of necessary resources, needing to allocate enough storage not only for their contained applications, but also for an entire guest operating system. In the context of servers still struggling with space and performance limitations, the use of a virtual machine in order to sandbox an application can therefore prove to be ineffective in terms of cost and resource consumption. On the other hand, encapsulating a server-side application together with all its dependencies ensures maximum portability and eases any potential future scaling. Container software provides a more lightweight alternative to the more traditional virtual machines, foregoing the use of a guest operating system for each instance in favor of sharing the kernel resources of the server's operating system across all of its contained applications.
More recently, the open-source project Docker has become almost synonymous with container software, gaining momentum and users in an almost exponential fashion. While container systems were available before the introduction of Docker, for example in the form of Linux Containers[34], Docker was the most successful at popularizing the concept. Figure 4.1 depicts the extreme increase in interest in the open source project since its release in March 2013:

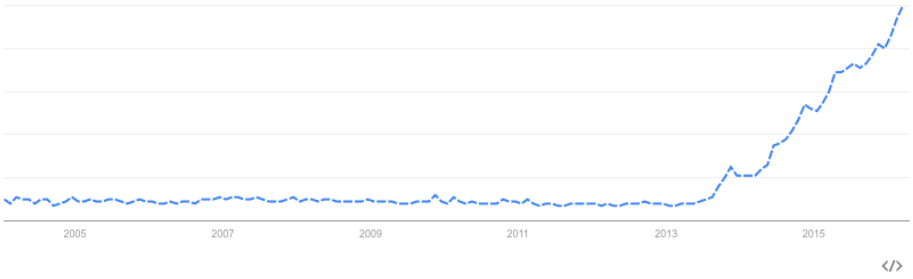
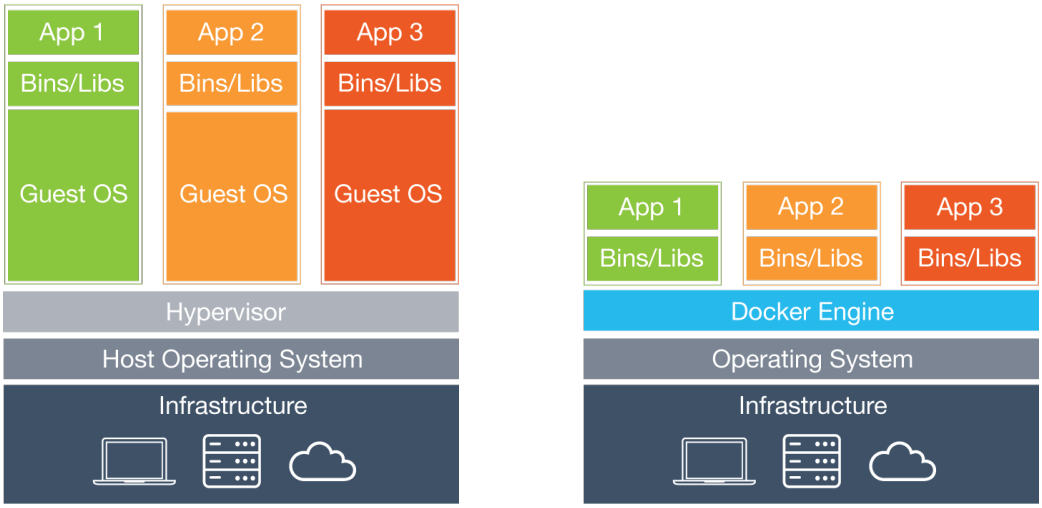


Figure 4.1: Google Trends graph depicting the interest in the term 'Docker'[35].

As opposed to virtual machines, which need to allocate enough resources for an entire guest operating system, as well as to simulate the hardware components of a physical machine, Docker container systems share the resources of the host's operating system and even the libraries that are common to more than one container instance. As can be seen in Figure 4.2, using Docker as a container system removes the need to create

[33] For a brief overview of the history of virtual machines, see: http://www.everythingvm.com/content/history-virtualization (visited on 30/06/2016)
[34] For more information about Linux containers, see *Linux Containers*. Canonical Ltd. June 30, 2016. URL: `https://linuxcontainers.org` (visited on 06/30/2016)
[35] Graphics source: https://www.google.com/trends/explore#q=%2Fm%2F0wkcjgj (visited on 30/06/2016)

and maintain instances of entire operating systems. As a result, Docker containers can be reduced in size from a few gigabytes to a few megabytes, depending on the size and number of dependencies of the encapsulated application.
On the other hand, applications encapsulated within virtual machines enable full isolation from any external perturbing factors, thus providing an additional layer of security which the Docker container software cannot yet guarantee.



(a) Architecture of a server hosting multiple virtual machines

(b) Architecture of a server hosting multiple Docker containers

Figure 4.2: The use of virtual machines and Docker containers to encapsulate applications[36]

Due to the fact that all implementations of the API should have the same underlying operating system, as stipulated in Chapter 3 as a requirement, as well as the fact that the security of the resulting applications was not defined as an analysis point for the upcoming evaluation, it was decided to use Docker containers in order to encapsulate each of the server-side applications. The use of Docker as a container software was also justified from the perspective of reducing resource use on the machine used for implementing and testing the API implementations, as well as by the ease with which contained applications can be managed.
The Docker software system itself is also based on the client-server architecture model. The Docker client is the component with which the user is required to interact in order to communicate with any Docker containers. In order to build, deploy or shutdown Docker containers, the user must specify the action to the Docker client, which then forwards the desired action to the Docker daemon, a process running on the host component of the Docker software system. The daemon process is then responsible for performing the operations requested by the client. Although the Docker client and the Docker host can reside on the same machine, as with any client-server architectures, they are conceptually separated and can therefore be deployed on sep-

[36] Graphics source: https://www.docker.com/what-docker (visited on 30/06/2016)

arate devices. Figure 4.3 depicts the separation of the client component from the host component, as well as the communication established between the two in order to manipulate any Docker containers responsible for running their corresponding embedded applications.
A further point to be considered is the use of images by the Docker software system. Docker images are read-only templates used for instantiating new Docker containers. A high number of open-source images are available for download in the Docker registry, with users having the additional possibility of creating and maintaining their own private Docker images based on Dockerfiles or already running Docker containers.

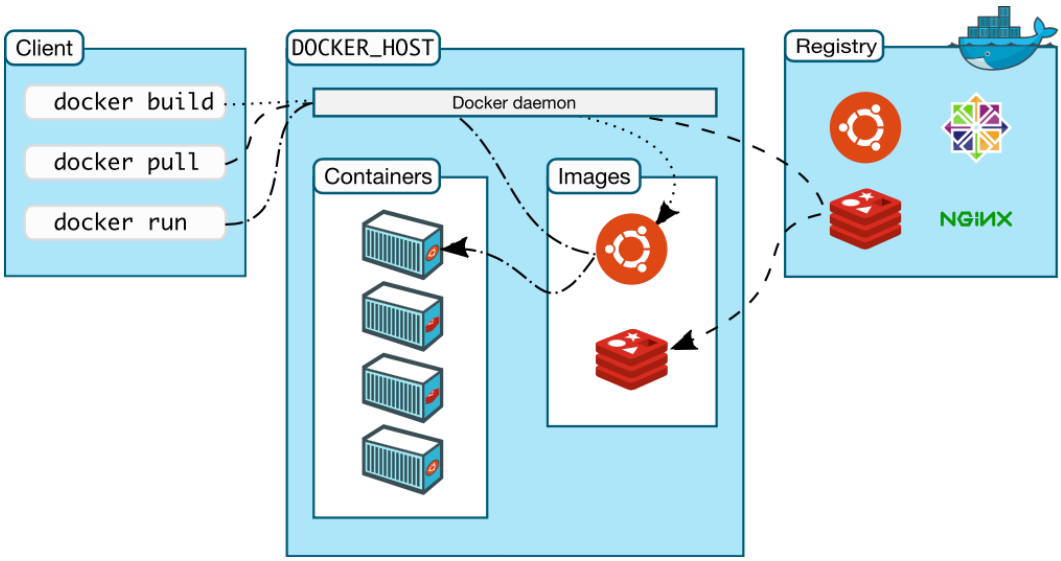


Figure 4.3: The architecture of a Docker system[37].

For the implementation and testing of the APIs, a machine based on the operating system OS X 10.11.3 was used. In the case of Linux-based machines, the `localhost` is also the Docker host. Thus, Docker containers can be addressed directly by using either the `localhost` hostname or the `0.0.0.0` IP address with their corresponding port number. However, due to the fact that Docker software is based on the Linux operating system, it cannot natively run in an OS X environment[38]. Instead, in the case of machines running an OS X operating system, a virtual machine based on Linux must first be created, which can subsequently be addressed by its corresponding IP address. This virtual machine will then include the Docker components normally running on the `localhost`, such as the Docker daemon and the containers, which can be addressed through the same ports by means of a one-to-one mapping to the ports

[37] Graphics source:
https://docs.docker.com/v1.11/engine/understanding-docker (visited on 30/06/2016)

[38] As of June 2016, a native Docker solution for the OS X environment has been released as a public beta version. As it was not yet an official production release at the time of the current paper's publication, it was not considered for the purpose of this thesis.

of the virtual machine. Figure 4.4 depicts how the Docker containers reside inside a Linux virtual machine, which is running in an OS X environment.

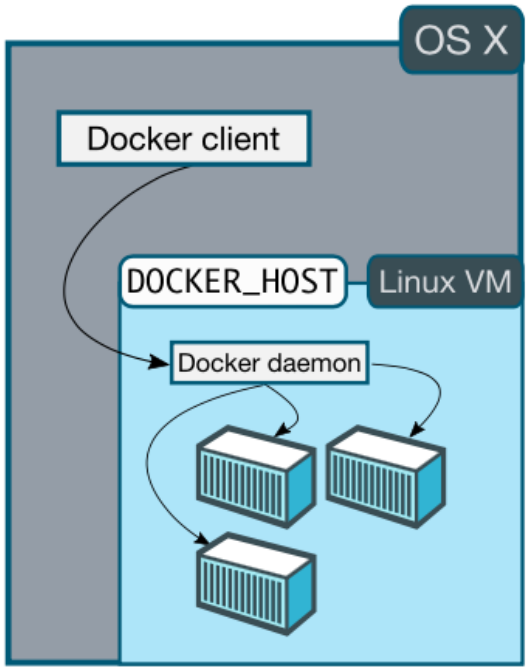


Figure 4.4: The architecture of a Docker system running in an OS X environment[39].

It is worth noting that although the Docker host is encapsulated in a Linux virtual machine, which in turn requires the use of more resources than by installing a native program, the overall resources required by this solution are still fewer than those which would be needed when implementing a solution based solely on virtual machines. Moreover, the use of Docker containers guarantees the possibility that future instances of the server components can be instantly ported to a Docker host running in a native Linux environment, thus ensuring maximum resource preservation.
As previously stated, the virtual machine housing the Docker host runs by default a Linux operating system, more specifically Linux 2.6 running in a 64-bit environment. The number of available CPUs is equal to that of the machine on which the virtual machine is running, namely four, while the amount of memory allocated was initially 4096 megabytes. While the operating system remained unchanged throughout the evaluation of the five contained server components, both the number of CPUs and the amount of available memory were changed according to varying test case specifications.
Based on the aforementioned Docker software container system, the server components of each of the implemented APIs will reside in a separate Docker container. All instances of the containers will be managed by the Docker daemon simultane-

[39] Graphics source: https://docs.docker.com/engine/installation/mac/ (visited on 30/06/2016)

ously, while the communication between the client component[40] and the containers will be carried out over the previously established network between the two machines. While a direct communication between the client application running on the Windows-based machine and the Docker containers running on the server machine would have been possible in the case of a Linux-based Docker implementation, the use of a machine running the OS X operating system prompts the automatic wrapping of the resulting containers in a Linux-based virtual machine. It is worth noting that the virtualization software used by the Docker system is VirtualBox[41], and that the established virtual machine receives by default an internal IP address, which can only be accessed from the host system itself, in this case from the machine running the server-side applications. It is therefore by default impossible to access the containerized server applications from the client component. In order to establish a successful communication path, it must first be ensured that the virtual machine containing the Docker server components be given an IP address valid for the entire network and not only for the machine on which it is running. Figure 4.5 demonstrates how to modify this default setting, namely by changing the default second network adapter of the virtual machine from a Host-Only Adapter to a Bridged Adapter over Ethernet. Once restarted with the newly configured settings, the virtual machine will receive an external IP address visible across the entire network, enabling communication between the client machine and the Docker containers.

[40] The current thesis uses the term `client` to refer to the client component of the RESTful API implementations and the term `Docker client` to refer to the client component of the Docker system responsible for communicating with the Docker daemon process.

[41] Further information about the software can be found under: https://www.virtualbox.org/wiki/VirtualBox (visited on 30/06/2016)

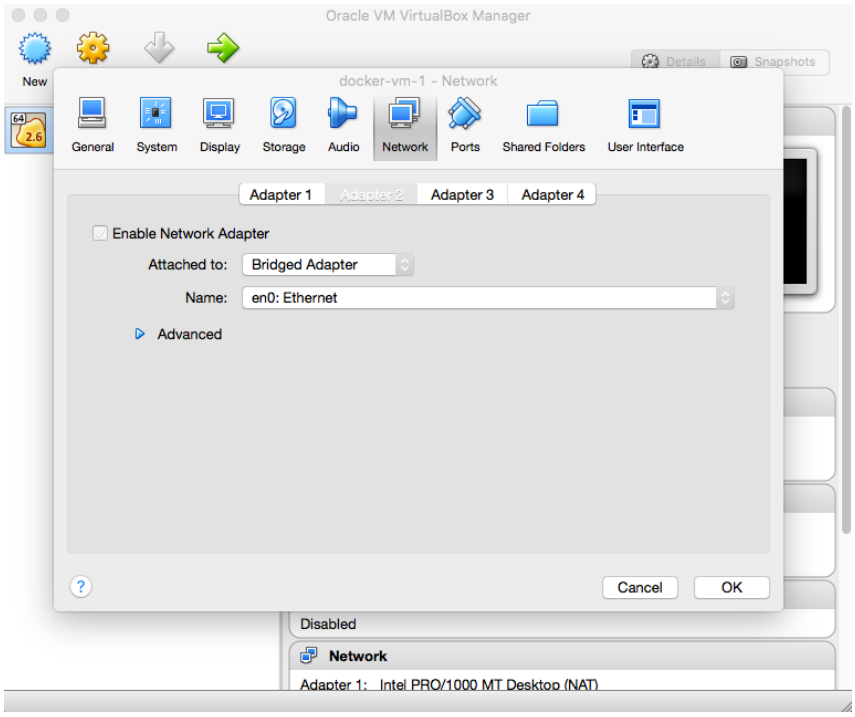


Figure 4.5: The network adapter configuration of the Docker system's VirtualBox in order to receive an IP address valid for the entire established network

Docker containers are isolated and secure application platforms containing everything needed by their corresponding application in order to run. Their life-cycle is governed by the Docker daemon process, as seen in Figure 4.4, and they are created based on Docker images. For example, a simple container based on the latest version of the Ubuntu operating system can be instantiated and run with the following command:

```
1 $ docker run -d -t ubuntu /bin/bash
```

The above command pulls the Docker image of the latest Ubuntu operating system, creates a container based on the image, creates an interface through which the container can communicate with the Docker host and executes a specified process within the container itself, in this case the simple bash process. The image specified in this example was that of an Ubuntu operating system and was pulled from the external Docker registry. However, the command can also be used with private images, which can be created either based on a running container, or based on a Dockerfile containing instructions for building an image. For the purpose of implementing the aforementioned APIs, Dockerfiles were written for each implementation. The contents of these Dockerfiles specified all the necessary steps for creating an isolated environment based on a Linux operating system and containing the server components of each RESTful service. The exact contents of the Dockerfiles will be detailed in the following sections of the current chapter.

Following the specification of each Dockerfile, the corresponding Docker images were

built, based on which the subsequent Docker containers were run. It is worth noting that Docker containers do not expose any ports by defaults, thus preventing any unwanted communication with external processes. In order to facilitate this communication and to control which port should be available to any incoming requests, the `run` command of the Docker client has an additional option used to map a container's port to the host. The `run` command used to start the containers therefore had the following additional option `-p hostPort:containerPort`, which was responsible for mapping the desired port of the Docker container to the Docker host. Different ports were specified for each implementation in order to facilitate the possibility of simultaneous communication between multiple clients and the server components. Addressing a specific application within a Docker container installed on the previously specified OS X operating system was therefore made possible by combining the IP address of the virtual machine with the previously exposed ports of each container respectively.

The use of the Docker container system is not only justified by its advantage when using and distributing resources on a machine with limited capabilities, but also from the perspective of testing the scalability of the different implementations, an analysis point specified in the previous chapter. Due to the fact that the machine used for the server-side applications runs on an OS X system, the Docker system is automatically contained in a virtual machine, whose hardware capabilities can be changed based on any test configuration requirements. Thus, in order to test for example the performance of a Swift-based server component when using a varying number of CPUs, no additional steps are required other than to change the configuration of the virtual machine itself and to shut down any other Docker containers that could compete for the same resources. The alternative strict use of virtual machines would not only require additional storage on the OS X machine itself, but also more time spent configuring the individual virtual machines for their respective test cases. Figure 4.6 shows how the VirtualBox software on which the Docker system is based can be used to configure the number of CPUs available to the Docker host running the server-side applications.

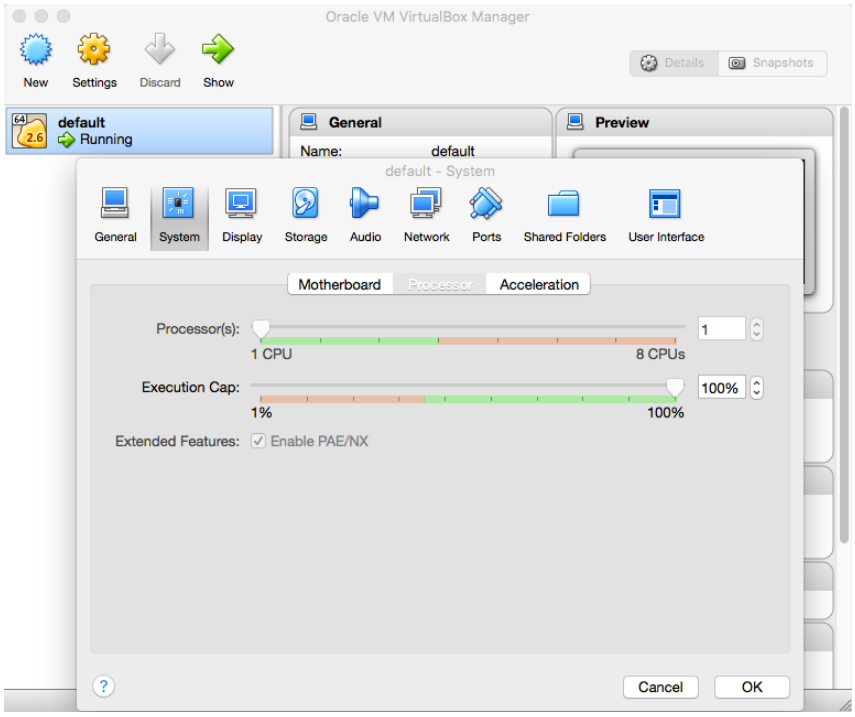


Figure 4.6: Settings panel of VirtualBox responsible for configuring the number of processors available to the Linux-based Docker virtual machine

The following sections illustrate how each API implementation was set up using the Docker container system, as well as how the underlying structure was achieved using each of the four technologies.

4.3 Node.js

As mentioned in Chapter 2, JavaScript can be used as a server-side language with the help of the Node.js framework, which uses an interpreter and is based on an event-driven architecture. The choice of using Node.js as opposed to other frameworks was based on the framework's popularity and ease of use. The Dockerfile used in order to configure the Node.js application contained the following instructions:

```
1  FROM node:0.10.40
2
3  WORKDIR "/usr/src/app"
4
5  ADD src/app.js app.js
6
7  RUN npm init -y && \
8      npm install express --save
9
10 EXPOSE 8888
11
```

```
12 CMD ["node", "app.js"]
```

The first line of the Dockerfile specifies the Docker image on which the container should be based, in this case the open source node:0.10.40 image found in the Docker registry. This image has a Debian Linux system as its foundation and is responsible for installing all the necessary dependencies in order to start implementing a Node.js application[42]. Among these dependencies is the node package manager, abbreviated `npm`, used to install and manage additional frameworks and libraries needed by a Node.js application.

The next line defines the working directory of the container to be the user's app directory, followed by the `ADD` command, responsible for copying the `app.js` file from the host system into the Docker container. It is worth noting that although this command requires the existence of a main application file outside of the container system, it does not represent a violation of the encapsulation principle upon which containers are based. Due to the fact that the Dockerfile is responsible for defining Docker images and not containers, its dependency on an external source is not only allowed, but often required. Once created based on its corresponding Dockerfile, an image will contain all its necessary resources and can be ported to any system. Likewise, a Docker container can subsequently be created and run based on this image, without the need to check for the availability of the required resources. Thus, encapsulation and portability are both maintained for the resulting Docker container. The content of the `app.js` file will be discussed in an ensuing paragraph of the current section.

Following the addition of external resources to the image, the `RUN` instruction was responsible for executing its subsequently specified commands in a shell program[43]. The first command was responsible for initializing a Node.js project and creating a `package.json` file containing its dependencies. In order to forego any necessary console interaction, the `-y` option was used, ensuring the use of default values wherever needed for the creation of the project. The most common way to implement a RESTful API with Node.js is by using one of its many open-source plugins, in this case the Express.js framework[44]. Therefore, the next command was used to install the framework and declare it as a dependency for the current Node.js project.

The following `EXPOSE` instruction informs the Docker host that the container will be listening on port 8888 at run time. This command does not ensure access for the Docker host to the container, but is necessary in order to establish a communication between the two. Mapping the internal container port to the external host is done

[42] The Dockerfile for the image itself can be found under: https://github.com/nodejs/docker-node/blob/b39ddbb7be87b9a2d1619f74757a5cec055c04ec/0.10/Dockerfile (visited on 30/06/2016)

[43] The documentation for the command can be found under: https://docs.docker.com/engine/reference/builder/#run (visited on 30/06/2016)

[44] Available and documented under *Express - Node.js web application framework*

by using the `-p` option in the corresponding `run` command used to start the Docker container.
Lastly, the `CMD` instruction specifies which application should be run once the Docker container is started. In this case, the `node` program is specified with the `app.js` argument, instructing Docker to start a Node.js server application based on the `app.js` file containing the following instructions:

```
1 var express = require('express');
2 var app = express();
3
4 app.get('/', function (req, res) {
5   res.set({
6     'Content-Type' : 'application/json',
7     'Cache-Control' : 'no-cache'
8   });
9   res.send(JSON.stringify({'response' : 'success', 'message': '
        ↪ successful message'}));
10 });
11
12 app.listen(8888, function () {
13   console.log('Application listening on port 8888!');
14 });
```

The file begins by including the `require` keyword[45] in order to load the `express` module and make it available in the current scope. An `express` application object is then created and assigned to the `app` variable. The `get` function is responsible for routing any HTTP GET requests made to a given pathway to their corresponding callback function. In this case, a simple GET request to the base URL of the application will trigger the defined function.
The callback function is specified to take two parameters, namely a request and a response object. The request object can be used in more detailed implementations to check for the existence of GET or POST parameters, as well as to get further information about the requesting user. In the case of the current implementation, however, the request object was ignored. The result object named `res` is used in order to send a response back to the client component. It first sets the response content to a JSON format and specifies that the response should not be cached by setting the appropriate HTTP headers. It then sends the previously defined JSON object through its `send` method. The readily available JavaScript method `stringify` of the `JSON` class is used to appropriately encode the response to be sent to the client. Lastly, the app is set to listen to any incoming connections on port 8888, corresponding to the port declared in the Dockerfile with the `EXPOSE` instruction.

[45] For an in depth explanation of the require and export functionality provided by Node.js, see Karl Seguin. *Node.js, Require and Exports*. Feb. 3, 2012. URL: `http://openmymind.net/2012/2/3/Node-Require-and-Exports/` (visited on 06/30/2016)

Based on the `app.js` file and the previously declared Dockerfile, the corresponding image can be created by using the following command:

```
1 docker build -t node .
```

The above `build` command creates a Docker image by the name of `node`, based on the Dockerfile found in the current directory. A Docker container running a Node.js application can then be started by using the following command:

```
1 docker run -d --name node -p 8888:8888 node
```

The `-d` option specifies that the container should be run in a detached mode[46] and should exit at the same time as the root process of the container. In the case of the current container, the root process is represented by the `node` process based on the `app.js` file and responsible for handling incoming requests from client components. The `-p` option is responsible for mapping the internal container port to the Docker host, in order to ensure the communication between the host and the container. Lastly, the command takes as input the previously created image named `node` and starts a container with the same name. Running the `docker ps -a` command will display all the Docker containers to be found in the Linux-based Docker virtual machine, including the recently created node-based container:

```
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS                    NAMES
219f8b432170        node                "node app.js"       6 seconds ago       Up 6 seconds        0.0.0.0:8888->8888/tcp   node
```

Figure 4.7: The description of a container running a Node.js application

Figure 4.7 shows that the node-based container is based on the Docker image with the same name, describes how its ports are mapped according to the previously used `run` command, as well as the uptime and creation time of the container itself. The `COMMAND` column shows the root process of the container, in the current case the `node` application responsible for listening to any incoming HTTP GET requests. According to the methodology defined in Chapter 3, the implementation of the application requires to be informally tested before any further tests can be carried out. By using the external Windows-based machine to address the application, a separation of the client and server components of the API is ensured. The server-side API implementation can only be accessed by specifying the IP address of the Docker host, as well as by communicating through the previously specified 8888 port. In order to send an HTTP GET request to the server component, the following command was used:

[46] The concept of a container running in a detached mode is explained in further detail under: https://docs.docker.com/engine/reference/run/#detached-d (visited on 30/06/2016)

```
curl 192.168.178.25:8888
```

cURL[47] is a command line tool used for transferring data with URL syntax and which supports HTTP GET requests. In the above command, the tool was used to send one request to the specified IP through the specified port. The IP address given as an argument is that of the Docker host and corresponds to the IP address of the Linux virtual machine encasing it. The response provided by the server was the following, in accordance to the format specified in Chapter 3:

```
{"response":"success","message":"successful message"}
```

By failing to specify the correct port, the following response is returned instead:

```
curl: (7) Failed to connect to 192.168.178.25 port 80: Connection
    ↪ refused
```

It can therefore be concluded that the server component based on Node.js is running correctly, is accessible to an external client through a previously specified entry point and returns a correctly formatted JSON response upon an HTTP GET request to the correct path.

4.4 PHP

As in the case of the previous section, the PHP implementation of the server component of the API was realized within a Docker container, ensuring the conceptual separation between the client and the server component. The Dockerfile used to configure the image on which the container was based had the following contents:

```
FROM php:5.6-apache

RUN apt-get update && \
    apt-get install -y git && \
    php -r "readfile('https://getcomposer.org/installer');" >
        ↪ composer-setup.php && \
    php -r "if (hash('SHA384', file_get_contents('composer-setup.
        ↪ php')) === '7228
        ↪ c001f88bee97506740ef0888240bd8a760b046ee16d
    b8f4095c0d8d525f2367663f22a46b48d072c816e7fe19959') { echo '
        ↪ Installer verified'; } else { echo 'Installer corrupt
```

[47] The full documentation of the command line tool can be found under *curl and libcurl*. June 30, 2016. URL: `https://curl.haxx.se` (visited on 06/30/2016)

```
            ↪ '; unlink('composer-setup.php'); } echo PHP_EOL;" &&
            ↪ \
8       php composer-setup.php && \
9       php -r "unlink('composer-setup.php');" && \
10      php composer.phar create-project slim/slim-skeleton php-rest &&
            ↪  \
11      cd php-rest
12
13  ADD src/index.php /var/www/html/php-rest/public/index.php
14  ADD run.sh /var/www/html/php-rest/run.sh
15
16  WORKDIR "/var/www/html/php-rest"
17
18  EXPOSE 8989
19
20  CMD ["sh", "run.sh"]
```

As with the previous Dockerfile, the `FROM` instruction determines which base image should be used for the construction of the container. In this case, the base image is php:5.6-apache, which has at its core a Debian Linux system and which is additionally responsible for configuring an Apache web server working in conjunction with PHP[48]. In order to facilitate the installation of the required libraries and frameworks, the `RUN` instruction was first used to install the PHP dependency manager named `composer`[49], similar to the `npm` package manager used in the aforementioned Node.js container. A commonly used framework for building RESTful APIs with PHP is the Slim micro framework[50], which was subsequently installed using the `composer` program. The last two shell commands provided to the `RUN` instruction were responsible for cloning a skeleton application, ready to use in order to implement a RESTful API.

The Dockerfile continues by copying the `index.php` file from the external OS X system into the container resources by using the `ADD` instruction. Additionally, the script file `run.sh` is added to the container, which includes the instructions necessary for starting the service. By changing the working directory to the newly created folder with the `WORKDIR` instruction, subsequent access to the files required to start the PHP service is simplified. Lastly, the port 8989 is exposed, enabling its later mapping to a host port.

As opposed to the previous Node.js container, the `CMD` instruction is not used directly to start the API service, but indirectly to run a script having this responsibility. The `run.sh` script contains the following instruction:

[48] The Dockerfile of the image itself can be found under: https://github.com/docker-library/php/blob/4677ca134fe48d20c820a19becb99198824d78e3/5.6/apache/Dockerfile (visited on 30/06/2016)

[49] The full documentation of the composer tool can be found under *Composer*. June 30, 2016. URL: `https://getcomposer.org` (visited on 06/30/2016)

[50] The full documentation of the framework can be found under *Slim Framework - Slim Framework*. June 30, 2016. URL: `http://www.slimframework.com` (visited on 06/30/2016)

```
#!/bin/bash
php -S 0.0.0.0:8989 -t public public/index.php
```

The above script file is responsible for starting a PHP service listening on port 8989 with its logic based on the contents of the `index.php` file added with the `ADD` instruction. The RESTful API itself is implemented using the Slim micro-framework in the `index.php` file as follows:

```
<?php
use \Psr\Http\Message\ServerRequestInterface as Request;
use \Psr\Http\Message\ResponseInterface as Response;

require 'vendor/autoload.php';

$app = new \Slim\App;
$app->get('/', function (Request $request, Response $response) {
  $newResponse = $response->withHeader('Content-type', 'application
        ↪ /json');
  $newResponse = $newResponse->withAddedHeader('Cache-control', 'no
        ↪ -cache');
  $newResponse = $newResponse->withJson(array('response' => '
        ↪ success', 'message' => 'successful message'));
  return $newResponse;
});
$app->run();
```

After including the necessary libraries and files, a new Slim application object is initialized and saved in the `app` variable. The behavior of the application when receiving an HTTP GET request is specified by using the `get` method of the object and by providing the general `/` path and a callback function. As in the case of the Node.js application based on the Express.js framework, the callback function is defined as having two parameters, namely a request object, also ignored in this implementation, and a response object. The response object first appends the necessary HTTP headers, namely the 'Content-type' header specifying that the client should expect a JSON-formatted response, and the 'Cache-control' header responsible for specifying the caching policy of the RESTful service. As the response object is immutable, a new variable must be initialized with the result of the previous method. Sending an appropriate response to the client consists in the case of the current implementation in creating an array object with the necessary data, converting it to a JSON format by using the `withJson` method of the response object and by returning the newly created response object to the client. Lastly, the application object is instructed to start listening on the port previously specified in the `run.sh` script by calling its `run` function.

As with the Node.js application, the image based on the Dockerfile was created by using the `build` command, while the actual container running the PHP-based RESTful API service was started using the `run` command with the corresponding port mapping:

```
1 docker run -d --name php -p 8989:8989 php
```

A subsequent use of the `docker ps -a` command lists two containers, namely the one running the previously created Node.js application, and the newly created container responsible for the PHP service, as can be seen in Figure 4.8:

```
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS                              NAMES
0126d7abe90f        php                 "sh run.sh"         29 minutes ago      Up 22 minutes       80/tcp, 0.0.0.0:8989->8989/tcp     php
219f8b432170        node                "node app.js"       23 hours ago        Up 23 hours         0.0.0.0:8888->8888/tcp             node
```

Figure 4.8: The list of two containers running server-side applications based on PHP and Node.js respectively

In addition to mapping its 8989 container port to the corresponding port of the Docker host, the Docker container running the PHP server application also exposes port 80. Due to the fact that the default port of an Apache web server is 80, the php:5.6-apache image on which the PHP container is based exposes this port by default. However, as the `-p` option of the `docker run` command only maps port 8989 to the Docker client, the PHP container will not be accessible over port 80.
Testing the PHP-based RESTful service with the cURL command line tool revealed that the API behaved as expected, returning the same JSON response as its Node.js-based predecessor, as well as returning error messages both when the incorrect port was specified, and when an incorrect URL path was requested.

4.5 Python

Following the previously established pattern, a third API server component based on Python was implemented using Docker containers. The Dockerfile which would be used as a subsequent base for the container running the application was composed in the following way:

```
1 FROM ubuntu:15.10
2
3 RUN apt-get update && \
4     apt-get install -y python python-pip && \
5     pip install Flask
6
7 WORKDIR '/root'
8
```

```
ADD src/run.py run.py

EXPOSE 5000

CMD ["python", "run.py"]
```

The Dockerfile begins as expected with the `FROM` instuction, specifying an Ubuntu Linux operating system as the base image for the API implementation. The `RUN` instruction is used in conjunction with a number of shell commands in order to install the `Python` runtime environment and a Python package manager called `pip` [51], corresponding to the Node.js `npm` and the PHP `composer` programs. The last shell command given to the `RUN` instruction is used to install an external Python micro framework named `Flask`[52] with the help of the previously mentioned package manager. As with the Express.js and the Slim frameworks, Flask was used in order to build and deploy a RESTful API. The choice to use it as opposed to other frameworks was based on its small size and on the ease of implementation it provides. After changing the working directory to the root folder with the `WORKDIR` instruction, the Python file `run.py` on which the RESTful web service was based was added to the file system of the Docker container. The `EXPOSE` command is used to establish the possibility of communication with a client component over port 5000. Lastly, the `CMD` instruction is responsible for starting a Python web service based on the contents of the `run.py` file.

The configuration of the RESTful service was realized in the `run.py` file as follows:

```
from flask import Flask, json, Response
app = Flask(__name__)

@app.route("/")
def getRequest():

    data = {
        'response'  : 'success',
        'message' : 'successful message'
    }

    response = Response(json.dumps(data))
    response.headers['Content-type'] = 'application/json'
    response.headers['Cache-control'] = 'no-cache'

    return response

app.run(host='0.0.0.0')
```

[51] A full documentation of the tool can be found under *pip 8.1.2 : Python Package Index*. June 30, 2016. URL: `https://pypi.python.org/pypi/pip` (visited on 06/30/2016)

[52] A full documentation of the framework can be found under *Flask (A Python Microframework)*. June 30, 2016. URL: `http://flask.pocoo.org` (visited on 06/30/2016)

Following an import instruction for the required libraries, an application object is initialized with the help of the Flask framework. The `route()` decorator[53] is used with the standard `/` path in order to define a trigger point for the following function `getRequest`. Upon a successful GET HTTP request to the specified pathway, the method will be called and will return the previously defined JSON response with the help of the additional `json` library. The response object will also include the correct HTTP headers that define its content type and its cache policy. Lastly, the `run` method of the application object is called in order to start the server. The default port on which the application will listen is 5000, corresponding to the port previously exposed in the Dockerfile. By specifying the host as an argument in the `run` method, the server is made publicly available to external sources.
The container image based on the Dockerfile is subsequently generated with the same `build` command as the previous Docker images, while the container itself running the application is started with the following `run` command:

```
1 docker run -d --name python -p 5000:5000 python
```

As can be seen, a container named `python` is started based on an image with the same name, mapping its internal port 5000 externally to the Docker host. Running the command `docker ps -a` reveals the following Docker containers:

```
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS                              NAMES
514b0da8fe1d        python              "python run.py"     15 minutes ago      Up 15 minutes       0.0.0.0:5000->5000/tcp             python
0126d7abe90f        php                 "sh run.sh"         44 hours ago        Up 44 hours         80/tcp, 0.0.0.0:8989->8989/tcp     php
219f8b432170        node                "node app.js"       2 days ago          Up 2 days           0.0.0.0:8888->8888/tcp             node
```

Figure 4.9: Three Docker containers running web services based on Node.js, PHP and Python

An informal test with the help of the cURL command line tool revealed that the Python-based Docker container was also running as expected, returning the expected JSON response when the standard `/` pathway was requested and a `404` error message when specifying any other path or resource. Specifying a port other than 5000 was also met with a connection refusal.

4.6 Swift

The fourth API implementation featured the newly released Swift language and interpreter as its underlying technology. The RESTful service was implemented following the previously established pattern of creating an isolated Docker container and configuring it to run a fully independent server-side API component responsible

[53] A detailed description of the concept of decorators in the Python language can be found under *PEP 318 – Decorators for Functions and Methods*. June 5, 2016. URL: `https://www.python.org/dev/peps/pep-0318/` (visited on 06/30/2016)

for handling GET requests and responding with an appropriately formatted JSON response. Due to the fact that the Swift language was released more recently compared to the rest of the aforementioned technologies[54], best practices for its use as a server-side technology have yet to be established. As a result, one cannot yet determine which available framework is best suited to build RESTful APIs, as opposed to the more seasoned previously mentioned technologies, where developer involvement and continuous implementation has led to a clear distinction between the available frameworks. In the case of the Swift language, however, it is necessary to take into consideration multiple frameworks when implementing the server component of the specified API, in order to eliminate the possibility that the underlying framework and not the technology itself is responsible for a potential loss in performance. It was therefore decided that two Swift-based server components of the API be built, namely one based on the Perfect library, and one based on the Frank framework.

4.6.1 Perfect-Library

The first implementation of the server component was based on the Perfect library. The Dockerfile containing the instructions necessary for configuring the container's image had the following contents:

```
1  FROM ubuntu:15.10
2
3  RUN apt-get update
4  RUN     apt-get install -y clang libicu-dev libssl-dev libevent-dev
           ↪ libsqlite3-dev libcurl4-openssl-dev uuid-dev make wget
           ↪ lsb-release
5  RUN     cd ~
6  RUN     wget -q -O - https://swift.org/keys/all-keys.asc | gpg --
           ↪ import -
7  RUN     gpg --keyserver hkp://pool.sks-keyservers.net --refresh-keys
           ↪  Swift
8  RUN     wget https://swift.org/builds/swift-2.2-release/ubuntu1510/
           ↪ swift-2.2-RELEASE/swift-2.2-RELEASE-ubuntu15.10.tar.gz
9  RUN     wget https://swift.org/builds/swift-2.2-release/ubuntu1510/
           ↪ swift-2.2-RELEASE/swift-2.2-RELEASE-ubuntu15.10.tar.gz.
           ↪ sig
10 RUN     gpg --verify swift-2.2-RELEASE-ubuntu15.10.tar.gz.sig
11 RUN     tar -xvzf swift-2.2-RELEASE-ubuntu15.10.tar.gz --directory /
           ↪  --strip-components=1
12 RUN     rm -rf swift-2.2-RELEASE-ubuntu15.10* /tmp/* /var/tmp/*
13
14 ENV PATH /usr/bin:$PATH
15
16 ADD Perfect /root/Perfect
```

[54] The Swift language was released as open-source in December 2015

```
17
18 WORKDIR "/root/Perfect/PerfectLib"
19
20 RUN    cd /root/Perfect/PerfectLib && \
21     make && \
22     make install
23
24 WORKDIR "/root/Perfect/API"
25
26 RUN    cd /root/Perfect/API/ && \
27     make && \
28     mkdir /root/Perfect/PerfectServer/PerfectLibraries && \
29     cp /root/Perfect/API/URLRouting.so /root/Perfect/PerfectServer/
           ↪ PerfectLibraries
30
31 WORKDIR "/root/Perfect/PerfectServer"
32 RUN    cd /root/Perfect/PerfectServer && \
33     make
34
35 WORKDIR "/root/Perfect/PerfectServer"
36
37 EXPOSE 8181
38
39 CMD ["./perfectserverhttp"]
```

As with the previous three implementations, the first instruction of the Dockerfile defines the image on which the resulting container should be based, in this case a Linux Ubuntu operating system. This specific version of Ubuntu was chosen based on Swift's current support for Linux systems, namely Ubuntu 15.10 and Ubuntu 14.04. An installation on a Debian Linux system was attempted, but was unsuccessful. The `RUN` instruction is then passed shell commands responsible for installing the necessary dependencies of the Swift environment[55], as well as the tools needed by the Perfect library[56], with the help of which the server-side RESTful service is built. Furthermore, the shell commands are responsible for downloading the archives containing the necessary Swift compiler and its related tools from their corresponding software repository, as well as for verifying their authenticity. It is worth noting that the Perfect library was compatible with the Swift 2.2 release version, but not with the Swift 3.0 development version. Compiling the project also produced compiler warnings due to the fact that several methods contained in the library itself were scheduled to become deprecated in a later release of the language.
Following the installation of the Swift 2.2 release version, the `Perfect` folder containing the Swift-based API implementation is added to the file system of the Docker container with the corresponding `ADD` command. A further `RUN` instruction is performed

[55] Specified under: https://swift.org/download/#using-downloads (visited on 30/06/2016)
[56] Available and documented under *Perfect: Server-Side Swift*. June 5, 2016. URL: `https://github.com/PerfectlySoft/Perfect` (visited on 06/30/2016)

in order to configure both the `Perfect` server and to compile the project, named `URLRouting`. Lastly, after changing the working directory to the folder containing the API implementation and exposing port 8181 in order to enable communication with the client, the Swift-based server is started by using the `perfectserverhttp` command. As with the previous three API implementations, an external library was used in order to expedite and simplify the development process of the server component. In the case of the current implementation, the Swift module named `Perfect` was used in order to create a server component listening on a particular port and responding to an HTTP GET request by sending the previously specified JSON response. This functionality was realized with the help of the last three `RUN` instructions of the Dockerfile, which performed the following steps after adding the `Perfect` project files to the file system of the Docker container:

1. The PerfectLib Swift module is built and installed using the `make` and `make install` shell commands. This module provides a set of core utilities that can be used for creating server-side Swift-based applications. For example, after its installation, the `perfectserverhttp` command becomes available for use, which will later be used in conjunction with the `CMD` Dockerfile instruction in order to start the server application.

2. The actual Swift-based server component is then compiled by using the `make` command inside the `API` folder, which contains the `PerfectHandlers.swift` implementation file. This file contains all the logic necessary for reacting to HTTP GET requests and will be detailed in the current section.

3. Following the build process, a new dependency folder is created for the server component, namely `Perfect Libraries` and the compiled `URLRouting.so` file is copied into it. Upon running the `perfectserverhttp` command, a server component will be started based on the files located in this folder. Thus, the compiled project must be copied into this folder prior to the building process of the server.

4. Lastly, the server component of the `Perfect` Swift module is built by using the `make` command.

As stated in the previous enumeration, the `PerfectHandlers.swift` contained the logic responsible for routing the HTTP GET requests as needed and had the following contents:

```
1 import PerfectLib
2
3 public func PerfectServerModuleInit() {
```

```
4
5   Routing.Handler.registerGlobally()
6     Routing.Routes["GET", "/"] = { _ in return GetRequestHandler()
            ↪ };
7 }
8
9 class GetRequestHandler: RequestHandler {
10
11   func handleRequest(request: WebRequest, response: WebResponse) {
12
13         let jsonEncoder = JSONEncoder()
14         do {
15             let responseString = try jsonEncoder.encode(["response"
                   ↪ : "success", "message": "successful message"
                   ↪ ])
16             response.appendBodyString(responseString)
17         }
18         catch {
19             response.setStatus(400, message: "Bad Request")
20         }
21         response.addHeader("Content-type", value: "application/json
                ↪ ");
22         response.addHeader("Cache-control", value: "no-cache");
23         response.requestCompletedCallback()
24   }
25 }
```

The file begins by importing the `PerfectLib` Swift module containing all classes and dependencies required in order to create a server-side component based on the `Perfect` library. It then continues with the definition of a public function named `PerfectServerModuleInit`, which all Perfect Server modules must expose. Upon running the server component with the `perfectserverhttp` command, the system will load the currently defined module and call the aforementioned function. The function is therefore the starting point of the entire server application and as such should be used in order to register handlers and perform one-time tasks. In the case of the current implementation, the `PerfectServerModuleInit` function was used in order to install the default routing handler and to define how a GET request to the default `/` pathway should be handled.

The `Routing` class has a default behavior when handling unknown GET requests, namely to send an appropriate `404` error message. In order to provide custom behavior, an instance of the `RequestHandler` class must be provided which specifies its own response. This functionality was achieved by creating the `GetRequestHandler` class, a subclass of the expected `RequestHandler` which provided its own implementation of the `handleRequest` base function, and by assigning an instance of it to the `Routes` attribute of the `Routing` class by means of a closure.

The `handleRequest` function itself is similar in construction and behavior to the call-

back functions defined for the previously mentioned three technologies. It takes a request and a response object as its parameters, the former of which is also not used. The `JSONEncoder` class is used in order to format the previously agreed upon JSON response. The `addHeader` methods are used in order to append the necessary HTTP headers to the response object sent back to the client. Finally, the response object receives the JSON-encoded structure as its content body and calls its `requestCompletedCallback` function. It is worth noting that the use of the catch block was not optional, but required by the compiler itself. In order to provide a safeguard against uncaught errors, the Swift compiler requires that all functions which could potentially throw errors be wrapped in catch blocks in order to stop their errors from propagating and crashing the entire application. In this case, the `encode` ↪ method of the `JSONEncoder` instance could produce an error and therefore needed to be appropriately handled. Upon its failure, the catch block would be executed instead, setting a status of `404` to the response object. Afterwards the execution of the application would continue as normal, with the response calling its completion callback function and the client receiving an appropriate message.
As with the previous server component implementations, the Docker image corresponding to the previously detailed Dockerfile was constructed using the `docker build` command and run with the following instruction:

```
1 docker run -d --name swift_perfect -p 8181:8181 swift_perfect
```

As can be seen in the above command, a Docker container named `swift_perfect` and based on the Docker image with the same name is started in a detached mode and maps the internal port 8181 of the container to the same port of the Docker host, in concordance to the port exposed in the Dockerfile of the container. Port 8181 was used in this case due to the fact that it is the default port on which the `Perfect` server component listens. A subsequent `docker ps -a` command reveals the following running Docker containers:

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                              NAMES
7e24e16539df        bc824eb40058        "./perfectserverhttp"    4 hours ago         Up 4 hours          0.0.0.0:8181->8181/tcp             swift_perfect
c954a4700c4a        node                "node app.js"            6 days ago          Up 6 hours          0.0.0.0:8888->8888/tcp             node
514b0da8fe1d        python              "python run.py"          13 days ago         Up 5 hours          0.0.0.0:5000->5000/tcp             python
0126d7abe90f        php                 "sh run.sh"              2 weeks ago         Up 6 hours          80/tcp, 0.0.0.0:8989->8989/tcp     php
```

Figure 4.10: Four Docker containers running web services based on Node.js, PHP, Python and Swift using the Perfect library

Lastly, an informal test of the server component was carried out with the help of the cURL command line tool by specifying the IP address of the Docker host and port 8181. The Swift-based Docker container was running as expected, returning the expected JSON-formatted response upon a GET request to the standard `/` pathway and an expected `404` response when trying to access a different path or resource.

4.6.2 Frank

The same server-side component was also built with the help of the Swift-based Frank library[57]. The Dockerfile responsible for configuring the corresponding container of the server component had the following contents:

```
1  FROM ubuntu:15.10
2
3  RUN apt-get update && \
4      apt-get install -y clang libicu-dev wget git && \
5      cd ~ && \
6      wget -q -O - https://swift.org/keys/all-keys.asc | gpg --import
           ↪  - && \
7      gpg --keyserver hkp://pool.sks-keyservers.net --refresh-keys
           ↪ Swift && \
8      wget https://swift.org/builds/development/ubuntu1510/swift-
           ↪ DEVELOPMENT-SNAPSHOT-2016-03-01-a/swift-DEVELOPMENT-
           ↪ SNAPSHOT-2016-03-01-a-ubuntu15.10.tar.gz && \
9      wget https://swift.org/builds/development/ubuntu1510/swift-
           ↪ DEVELOPMENT-SNAPSHOT-2016-03-01-a/swift-DEVELOPMENT-
           ↪ SNAPSHOT-2016-03-01-a-ubuntu15.10.tar.gz.sig && \
10     gpg --verify swift-DEVELOPMENT-SNAPSHOT-2016-03-01-a-ubuntu15
           ↪ .10.tar.gz.sig && \
11     tar -xvzf swift-DEVELOPMENT-SNAPSHOT-2016-03-01-a-ubuntu15.10.
           ↪ tar.gz --directory / --strip-components=1 && \
12     rm -rf swift-DEVELOPMENT-SNAPSHOT-2016-03-01-a-ubuntu15.10* /
           ↪ tmp/* /var/tmp/*
13
14 ADD Frank /root/Frank
15
16 RUN cd /root/Frank && \
17     swift build
18
19 WORKDIR "/root/Frank"
20
21 EXPOSE 8000
22
23 CMD [".build/debug/Swift_Frank"]
```

As in the case of the previous Swift implementation, an Ubuntu version 15.10 is used as a base image. In the case of the `Frank` framework, a Swift package manager was used to manage dependencies to external Swift modules, not unlike the previous implementations using Node.js, PHP and Python. The Swift Package Manager[58] is integrated with the Swift build system and can be accessed after a successful installation of the environment. However, it only became available with version 3.0 of

[57] Available and documented under *Frank*. June 5, 2016. URL: `https://github.com/nestproject/Frank` (visited on 06/30/2016)

[58] Documented under: https://swift.org/package-manager/ (visited on 30/06/2016)

Swift, prompting the installation of a different Swift archive in the `RUN` instruction of the Dockerfile, as can be seen in the previous code example. Furthermore, despite continuous improvements and updates, the Frank project is also based on a slightly older version of Swift than the most recently available, building successfully with the development version 3.0 released on the 1st of March, 2016, but failing to compile with the newer 3.0 version released on 12th of April, 2016.
Following the download, installation and unpacking of the Swift tool ecosystem, the folder `Frank` containing the source code for the server application was added to the file system of the Docker container. Running the `swift build` command inside the newly added folder created an executable file with the same name which was responsible for starting a process listening on port 8000 for incoming `GET` requests. The `EXPOSE` instruction was used to ensure the possibility of communication between the aforementioned listening process and the Docker host.
The Swift Package Manager responsible for managing dependencies expects a `Package.swift` file containing a specification of its corresponding projects, along with the libraries which should be included in it. In the case of the current project, the file had the following contents:

```
1 import PackageDescription
2
3 let package = Package(
4   name: "Swift_Frank",
5   dependencies: [
6     .Package(url: "https://github.com/nestproject/Frank.git",
          ↪ majorVersion: 0, minor: 3),
7   ]
8 )
```

After importing a framework required by the `swift build` tool, the file defined a package constant having the name `Swift_Frank` and an array of dependencies. It is worth noting that the project was unable to be be compiled with its original name `Frank`, due to naming conflicts between the imported framework and the actual project. In order to properly function, the Swift Package Manager also expects a directory named `Sources` containing all the source files of the application, in this case the `main.swift` file having the following contents:

```
1 import Frank
2 import Inquiline
3
4 get { request in
5     var response = Response(.Ok, contentType: "application/json",
          ↪ content: "{\"response\": \"success\", \"message\": \"
          ↪ successful message\"}")
```

```
6       response["Cache-Control"] = "no-cache"
7       return response
8   }
```

Similarly to the previous three implementations using Node.js, PHP and Python, the file first imports the necessary libraries and then specifies how a GET request to the default `/` pathway should be handled. The `Frank` library is required in order to be able to define this behavior, while the `Inquiline` library represents a secondary dependency installed alongside `Frank` necessary for accessing the `Response` structure definition. As can be seen in the previous code example, a GET request to the default pathway is defined as a function taking a request object as a parameter and returning a corresponding response object. What is noteworthy in this implementation is the use of closure syntax, as opposed to the previously used full-bodied functions. As with the previous implementations, the function sends the expected JSON-formatted response upon a successful GET request to the default pathway with the appropriate HTTP headers. It is worth noting that the use of the NSJSONEncoder class was attempted in order to pass a true JSON object back to the requesting client. However, this class is only available within the Foundation framework[59], a standard Objective-C framework that has not yet been included within the Swift tool environment.
Running the `swift build` command in the root folder of the project produces an executable file with the same name, which upon running starts a server application listening on the default port 8000. The Docker image corresponding to the previously defined Dockerfile was constructed with the `docker build` command and run according to the established pattern with the following command:

```
1 docker run -d --name swift_frank -p 8000:8000 swift_frank
```

The above Docker command starts a container by the name of `swift_frank` in detached mode, based on the image with the same name and mapping port 8000 of the container to its corresponding port on the Docker host. A further `docker ps -a` command confirms the successful implementation of the Docker container:

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                              NAMES
0504eb2fccb3        swift_frank         ".build/debug/Swift_F"   17 hours ago        Up 17 hours         0.0.0.0:8000->8000/tcp             swift_frank
7e24e16539df        bc824eb40058        "./perfectserverhttp"    23 hours ago        Up 22 hours         0.0.0.0:8181->8181/tcp             swift_perfect
c954a4700c4a        node                "node app.js"            6 days ago          Up 24 hours         0.0.0.0:8888->8888/tcp             node
514b0da8fe1d        python              "python run.py"          13 days ago         Up 23 hours         0.0.0.0:5000->5000/tcp             python
0126d7abe90f        php                 "sh run.sh"              2 weeks ago         Up 24 hours         80/tcp, 0.0.0.0:8989->8989/tcp     php
```

Figure 4.11: Docker containers running web services based on Node.js, PHP, Python and Swift using the Perfect library and the Frank framework respectively

Finally, the correct functioning of the server-side component was tested by using

[59] Documented under:
https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/ObjC_classic/
(visited on 30/06/2016)

the cURL command line tool with the IP of the Docker host and the previously specified port 8000. As expected, the application returned the correct JSON response when receiving an HTTP GET request at the default `/` pathway. A request to an nonexistent resource or path, however, was met with an empty response on the client side.

4.7 Client implementation

Due to the fact that the machine designated for hosting the client component of the API was running a Windows operating system, a compatible application was required in order to perform any performance tests of the five server components. As Java-based software is inherently cross-platform compatible, the Apache JMeter[60] open-source and pure Java application was chosen as a testing tool. According to its official documentation, the application is specifically designed to test performance on a variety of static and dynamic services, including RESTful web services based on HTTP. It can be used to simulate heavy load on server components and to analyze the overall performance of various services.
The first step in configuring the tool was to confirm that its required dependencies were installed on the machine, namely a fully compliant JVM[61] version 6 or higher. The JMeter tool could then be downloaded and installed onto the machine without any further configuration being required. The graphical user interface of the application could be started from the command line by running the `jmeter.bat` file.
In order to prepare the tool for its use in the next phase of the analysis, a test configuration was established. Based on the guides provided by the documentation itself[62], a standard web test plan was developed in order to connect to the five server components. After creating a new test plan template, a Thread Group element was added to its configuration. It was configured to create 100 different users concurrently, as well as to repeat the test phase 5 times. Figure 4.12 depicts how the aforementioned thread group was configured. The ramp-up period determines the period of time over which the users should be created, a value of 0 dictating that they should all be created at the same time.

[60] Available and documented under *Apache JMeter*. June 5, 2016. URL: `http://jmeter.apache.org` (visited on 06/30/2016)
[61] Java Virtual Machine (JVM)
[62] Available under:
http://jmeter.apache.org/usermanual/build-web-test-plan.html (visited on 30/06/2016)

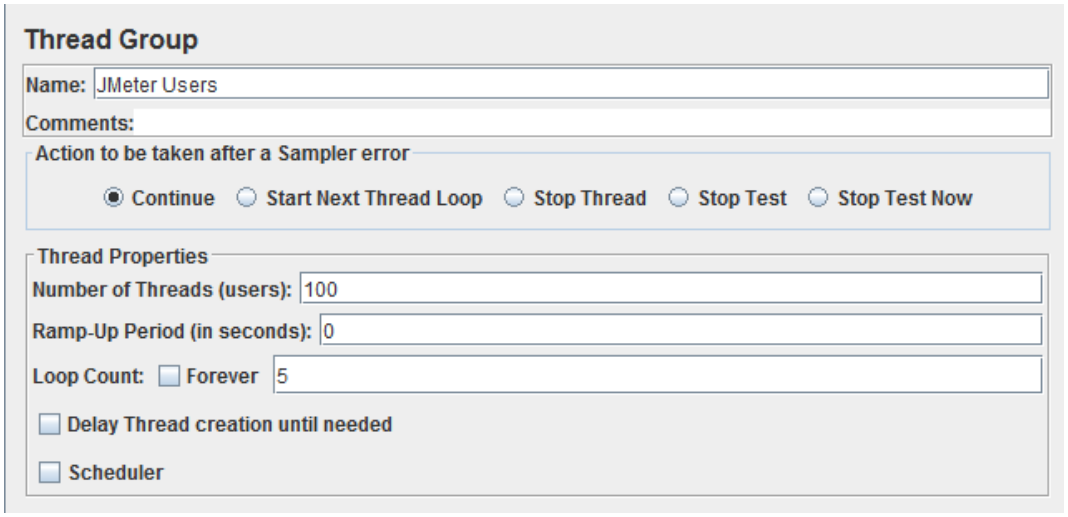


Figure 4.12: The configuration of the thread group for the initial JMeter test plan

The next step was to specify the server's information, namely its IP address and the port over which the connection should be made. In order to achieve this, a new object of the HTTP Request Defaults type was created and assigned to the thread group, specifying the external IP of the Docker host, equivalent to that of the Linux virtual machine, and port 8181, over which one of the Swift-based server components could be reached. Figure 4.13 depicts the above specified configuration:

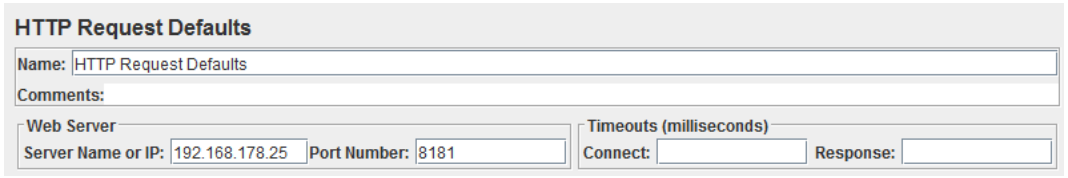


Figure 4.13: The configuration of the HTTP Request Defaults object for the initial JMeter test plan

Although both the IP address and the port of the server component were specified, the JMeter tool additionally requires an explicit specification of which requests to make to this address. Therefore, an HTTP Request object was created and assigned to the previously defined Thread Group. It was configured to make a simple HTTP GET request to the standard `/` pathway, as depicted in Figure 4.14. As can be seen in the following figure, both the IP address and the port number fields can remain empty due to the fact that their default values were previously specified in the HTTP Request Defaults object:

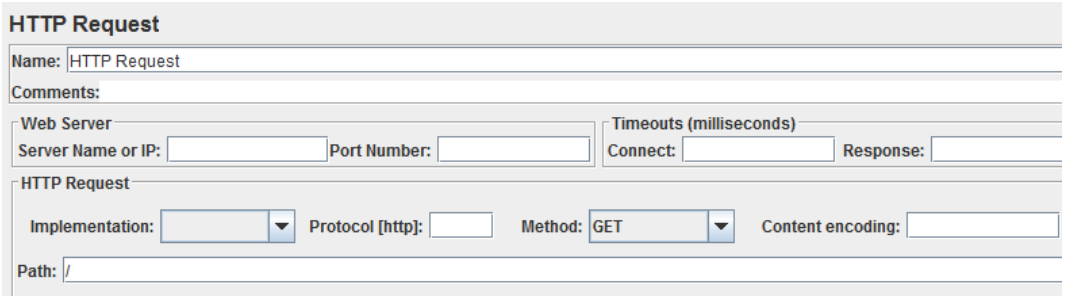
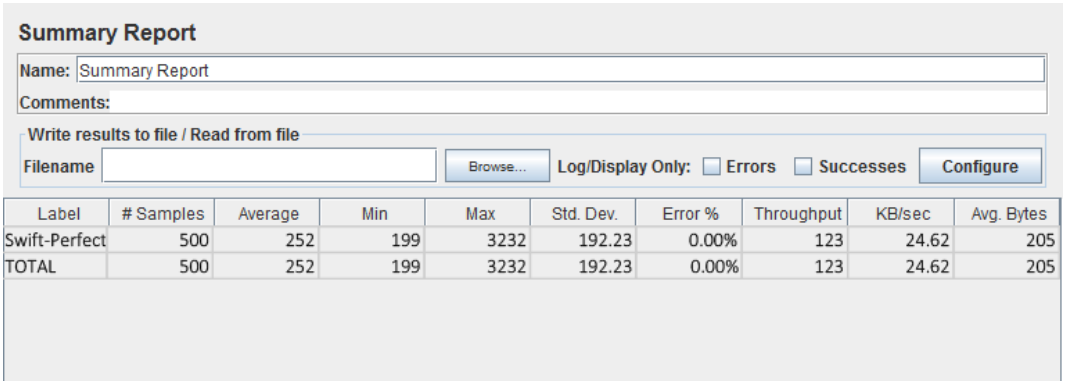


Figure 4.14: The configuration of the HTTP Request Defaults object for the initial JMeter test plan

Lastly, a Listener object was created and assigned to the Thread Group, in order to store the results of the performed test. A Summary Report object was chosen based on the data it recorded, namely average response time and error rate, as well as due to the fact that it consumed constant memory, as opposed to its Aggregate Report alternative. Upon running the specified test, the following output was obtained:



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
Swift-Perfect	500	252	199	3232	192.23	0.00%	123	24.62	205
TOTAL	500	252	199	3232	192.23	0.00%	123	24.62	205

Figure 4.15: The output provided by the previously defined test plan when accessing one of the Swift-based server components over a GET request

As can be seen from the above figure, the provided output includes an average response time of all performed requests, as well as an error rate provided in percentage form. The Summary Report object could therefore be used in order to evaluate both the performance and the reliability of the five API implementations.

4.8 Test configuration

In order to test the performance, reliability and scalability of the five server components, a software capable of sending multiple requests in a predefined time frame and storing the corresponding response statistics was needed. Based on these requirements, the JMeter software was chosen as an appropriate testing tool. The client component, as required by the previously specified methodology, was shared across all API implementations in order to avoid any unnecessary discrepancies between evaluation results.

A Thread Group object was defined for each of the five API implementations, with its corresponding HTTP Request Defaults, HTTP Request and Summary Report objects. Each HTTP Request Defaults object contained the IP address of the external machine hosting the server applications, while the HTTP Request object specified that a GET request should be made to the default `/` pathway. The five HTTP Request objects differed solely in their specified port, which corresponded to the ports forwarded by the virtual machine running the Docker host, as well as to the internal ports on which the docker containers were listening. Thus, the Node Thread Group was configured to send multiple GET requests to port 8888, while the Swift-Perfect Thread Group had port 8181 assigned to it. The resulting test plan had the following structure:

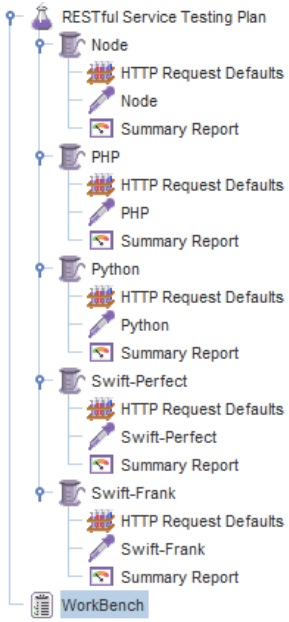


Figure 4.16: The structure of the JMeter test plan used in the evaluation phase

In order to ensure that the server components were allocated the entirety of the virtual machine's memory, only one docker container was allowed to run during the execution of its own tests. At the same time, the client side ensured that every test be executed separately and successively by toggling the online status of each of the Thread Group objects. For example, testing the server component based on the Swift language and using Frank as the underlying framework entailed the pausing of all docker containers except for `swift-frank` and the disabling of all JMeter Thread Group objects expect the one responsible for sending and recording GET requests through port 8000, namely `Swift-Frank`.

The configuration of the number of requests, as well as the number of times the test should be repeated, was done by filling out the Number of Threads and the Loop Count fields in each Thread Group object with the desired values. In the case of the

performed evaluation, each test was repeated five times, in order to obtain a reliable median value of the monitored API analysis points.
In order to measure the performance of the five implementations, 480 different tests were performed, varying in the number of requests, number of allocated CPUs for the virtual machine hosting the server components, and the amount of memory available to the aforementioned virtual machine. Tests were performed with 10, 100, 500, 1000, 1500 and 2000 concurrent threads and a virtual machine having 1024MB, 2048MB, 3072MB and 4096MB of allocated memory respectively, as well as a number of CPUs ranging between one and four. Tests consisting of each combination of the above specified configurations were performed for each of the API implementations five total times and the average response times and error rates were recorded by the corresponding Summary Report objects. Lastly, the raw data was exported into individual .csv files and aggregated into corresponding graphical representations.

5 Evaluation

In order to properly evaluate the five server components based on their underlying technologies, a set of analysis points was defined in Chapter 3, namely functionality, performance, reliability, scalability, size and ease of implementation. The following chapter describes the evaluation process of each of the API implementations based on the aforementioned analysis points, as well as the results of this process.

5.1 Compliance to API requirements

Prior to a formal analysis of the previously described implementations, their conformance to the requirements defined in Chapter 3 must be formally verified. The first requirement for each of the implementations was their compliance to the RESTful software architectural style.The following section will justify how each of the previously defined criteria for a RESTful service was met by all five implementations:

1. Client-server
   All five implementations of the service consisted of a shared client component and a server component separated not only on a software, but also on a hardware level. The client component was responsible for requesting resources from the server components through a previously defined interface, without having additional insight into the internal logic of the implementation required for providing an appropriate response. While the client component was shared by all five implementations, each server component coupled with the client instance can be considered an individual and separate service. Thus, the client-server requirement of the RESTful software architectural style was met by all five implementations.

2. Stateless
   The underlying protocol used for enabling the communication between the client and the server component of each of the five implementations was HTTP. Thus, each request made by the client was treated by default by the server as independent from any other previous requests, ensuring the conformance to the stateless requirement of RESTful services.

3. Cacheable
   Each of the five server implementations had the possibility to specify the caching behavior of the client by setting the corresponding `Cache-control` HTTP

header to a particular value, in this case to `no-cache`. Thus, each implementation had the possibility to notify the client of any potentially cacheable responses, ensuring the services' conformance to the corresponding requirement.

4. Layered system
   The layered system constraint of the RESTful architectural style stipulates that each architectural layer be unaware of any other layers not immediately adjacent to itself. Due to the fact that the server components and the client implementation each consisted of only one layer, this requirement was met without any need for further adjustments.

5. Uniform interface
   Meeting the uniform interface requirement was ensured by the conformance to the following four constraints:

   a) Identification of resources
      All five implementations of the service had access to a single resource, namely the previously defined JSON object. As this object could be uniquely identified by using the default `/` pathway prepended by the server component's IP address, all five services conformed to the identification of resources requirement.

   b) Manipulation of resources through representations
      Due to the fact that the provided JSON resource was read-only, ensuring its availability at the default `/` pathway suffices in order to conform to the aforementioned criterion.

   c) Self-descriptive messages
      Alongside the `Cache-control` HTTP header, a further `Content-type` header was used in order to alert the client as to the type of response it should expect, namely an `application/json` response. Thus, the response provided by each of the server components contained enough information in order to ensure its correct parsing by the client component.

   d) Hypermedia as the engine of application state (HATEOAS)
      As the defined structure of the API consisted of only one entry point, namely the default `/` pathway, all five service implementations formally conform to the HATEOAS principle.

Alongside the formal constraints of the RESTful architectural style, a set of additional constraints was specified for the five implementations:

- Identical operating system and resources
  All five server components were implemented as Docker containers based on a Linux operating system in the same Docker virtual machine, thus ensuring their access to the same software resources. In order to ensure that each application

was also identical from a hardware perspective, only one server component was allowed to run at a time, ensuring that all hardware resources of the virtual machine were allocated to the running container and not being spent on background processes pertaining to the management of other services.

- Communication over HTTP
  All five implementations were configured to communicate over the HTTP protocol by using appropriate frameworks on the server side and by sending HTTP requests from the client component.

- One HTTP service call
  Each of the five server components exposed a single service call at the default `/` pathway. This service was also only made available over an HTTP GET request.

- JSON response
  All five implementations returned the same response encoded in JSON format and conforming to the previously defined structure.

- Identical client component
  The same client component was used in order to communicate with all five of the server applications, namely the JMeter application running on a separate Windows machine.

- Separate machines
  As described in an earlier section of the current chapter, one machine was used in order to host the Docker virtual machine containing the server components, while a different machine was used to house the JMeter client component. The communication between the two components was subsequently enabled by connecting the two machines to the same network and making the processes visible to each other with appropriate IP addresses.

The current section has illustrated that all five API implementations met the requirements specified in Chapter 3 and could therefore proceed to the evaluation phase of their analysis.

5.2 Functionality

The ISO/IEC 25010 norm[63] defines functionality among its six external and internal software quality characteristics and illustrates it as consisting of the following five attributes[64]:

[63] An international standard used for the evaluation of software quality

[64] As specified in *ISO/IEC 25010:2011(en) Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models.* 2011, Chapter 4.2

1. Suitability
   A software system is considered suitable if it provides functions that meet the stated and implied needs when used under specified conditions. In the case of the current thesis, all five API implementations provide the necessary functions that meet the needs specified in Chapter 3, namely to provide a JSON-encoded response upon receiving an HTTP GET request at the default `/` pathway. This behavior was tested with the help of the cURL command line tool and described throughout Chapter 4 at the end of each implementation section. It can therefore be concluded that all five API implementations are suitable.

2. Accuracy
   A software system is considered to be accurate if and only if its functions return a correct output upon receiving an acceptable input value. All five API implementations delivered the same JSON-encoded response specified in Chapter 3 upon receiving an appropriate HTTP GET request at the default pathway. All five implementations can therefore be considered accurate.

3. Interoperability
   This sub-characteristic describes the ability of a software system to communicate with other software components. By conforming to the RESTful architectural style and providing a uniform interface in order to facilitate the communication with any potential client component, all five API server component implementations support interoperability.

4. Compliance
   This sub-characteristic addresses the capability of a software system to be compliant with necessary laws and guidelines. Due to the fact that all five applications were implemented solely for research purposes and were not intended for release, this point becomes irrelevant in the analysis of the server components' functionality.

5. Security
   This sub-characteristic relates to unauthorized access to the functions provided by the software system. The sole resource exposed by all five of the implementations is made available through a single API entry point, namely the `/` pathway through an HTTP GET request. Any request differing in either type or pathway is met with an expected 404 response. Thus, it can be concluded that the five API implementations also deliver the minimum expected security requirements.

As all five API server components comply to the aforementioned sub-characteristics of the functionality software quality characteristic specified by the ISO/IEC 25010

norm, it can be concluded that all five implementations are fully functional. Moreover, all five server components are identical in their degree of meeting the previously specified criteria for functionality.

5.3 Disk usage

The size of each of the docker containers can be displayed using the command `docker ps -s`, which lists all running docker containers with their corresponding disk and virtual sizes:

```
NAMES               SIZE
swift-frank         0 B (virtual 1.044 GB)
swift-perfect       3.072 kB (virtual 1.17 GB)
python              0 B (virtual 392.3 MB)
php                 0 B (virtual 548.7 MB)
node                0 B (virtual 636.3 MB)
```

Figure 5.1: All five docker containers with their corresponding sizes

As can be seen in the above figure, the least costly implementation from the perspective of disk usage uses Python as its underlying technology, with PHP and Node.js following closely behind. On the other hand, both Swift-based server implementations needed at least 1GB of disk space, rendering Swift the most disadvantageous technology for implementing server-side applications from the perspective of required storage capacity.

5.4 Ease of implementation

In order to objectively evaluate the ease with which the five server components were implemented, criteria such as developer experience were excluded from the analysis. Therefore, the first aspect considered in the evaluation of the APIs' ease of implementation was the number of lines of code required to achieve the same functionality. Configuration files, as well as external libraries, were not considered in this evaluation. The minimum number of lines of code needed to implement the API server component was 8 and was achieved by the Swift-based API having Frank as its underlying framework. The Node.js and PHP-based implementations followed with a number of 14 lines of code, succeeded by 18 lines required by the Python-based component. Lastly, the Swift-based implementation having the Perfect library as its underlying framework required 25 lines of code. The difference between the minimum and the maximum number of lines of code needed to implement the same service is small enough to be considered insignificant. The five technologies used to implement the APIs can therefore be considered equivalent from the perspective of required lines of code.

An additional point to be considered was the number of steps necessary to configure the environment needed by the applications. The Dockerfile needed to configure the Node-based application consisted of 17 lines of code and was responsible for initializing a Node.js project and starting it. The Dockerfile for the Python-based implementation consisted of 19 lines of code and was responsible for installing the Python interpreter and package manager, as well as for installing the Flask library required by the implementation itself. The Dockerfile for the PHP implementation also consisted of only 21 lines of code responsible for installing the `git` program and the `composer` package manager, as well as for creating a new Slim-based project and starting the resulting service. Lastly, the Dockerfiles corresponding to the Swift-based implementations displayed a higher number of lines of code, namely 24 for the Frank-based implementation and 43 for the Perfect-based component. Both were responsible for downloading the Swift packages and the programs on which they are dependent, as well as to verify the validity of the downloaded software. While the Frank-based project subsequently needs only to be compiled, the Perfect-based component first requires compiling the necessary Perfect libraries, linking the API implementation to the project and lastly building the server component before the service can be started. A notable point to be considered was the necessity to use different versions of the Swift package for Ubuntu based on the underlying technology. While the Frank-based project would only work with Swift 3.0 or above, the Perfect library was only compatible with Swift 2.2. Due to the fact that the official Docker image repository does not yet contain any Ubuntu 15.10 images responsible for configuring a Swift environment, the Swift-based containers proved to be the most difficult and time-consuming to set up. Of the two, the Perfect-based Swift implementation required the most effort due to the additional need to compile multiple project components before being able to run the service.

5.5 Efficiency

In terms of software quality, the efficiency of a software system is defined by the ISO/IEC 25010 norm as the capability to provide appropriate performance relative to the amount of resources used, under predefined conditions. It is described as consisting of three sub-characteristics, namely:

1. Time behavior
   This sub-characteristic refers to the capability of the software system to provide appropriate response and processing times as well as throughput rates when performing its functions, under predetermined conditions. In the case of the current API server components, the time behavior was evaluated and compared by measuring the average response times of each of the five implementations under optimal conditions and while handling varying number of concurrent requests.

2. Resource Utilization
   This sub-characteristic evaluates the capability of a software system to use appropriate amounts and types of resources when performing its functions under predetermined conditions. In the case of the current analysis, the host system on which the five implementations were running was identical, namely a Docker virtual machine. It could therefore be possible to observe the behavior of each implementation when confronted with varying memory and a variable number of allocated CPUs.

3. Efficiency compliance
   This sub-characteristic refers to the capability of the software system to adhere to standards or conventions relating to efficiency. Due to the fact that the five server components were implemented solely for research purposes, this point was not taken into consideration when comparing the five server-side applications.

The following subsections will provide an insight into the evaluation of the time behavior and resource utilization of the five API implementations.

5.5.1 Time behavior

In order to evaluate the time behavior of each of the five API implementations, an optimal hardware configuration was chosen for all tests, namely an allocated 4096MB of memory and four CPUs to the Docker virtual machine hosting the server components. At the same time, only one Docker container was allowed to run at a time in order to exclude the competition for resources as a potential influencing factor in the analysis. On the client side, six different tests were performed with a number of concurrent threads of 10, 100, 500, 1000, 1500 and 2000 respectively. Each test was performed five times for each API implementation. The resulting average response times were subsequently gathered into a single file and displayed graphically based on their underlying technologies, as can be seen in the following figure:

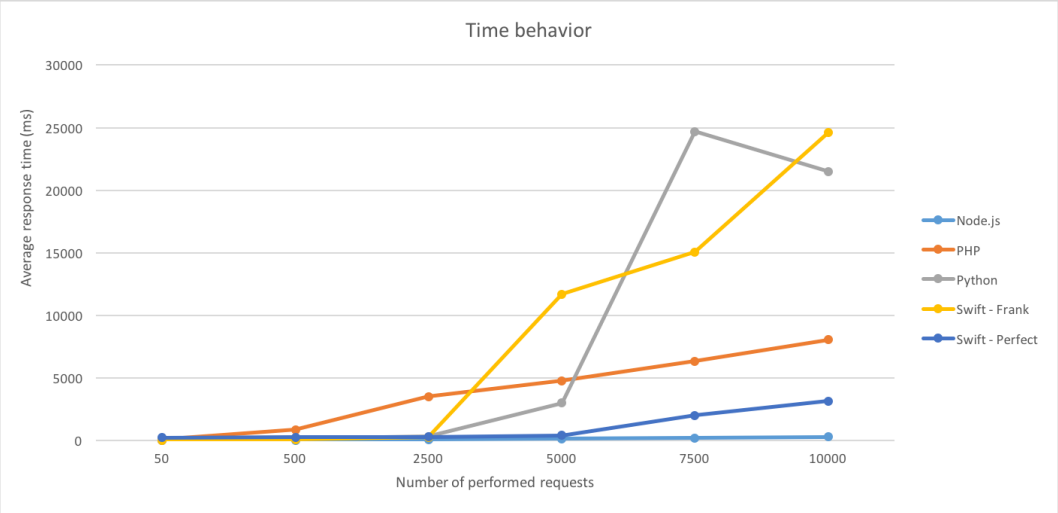
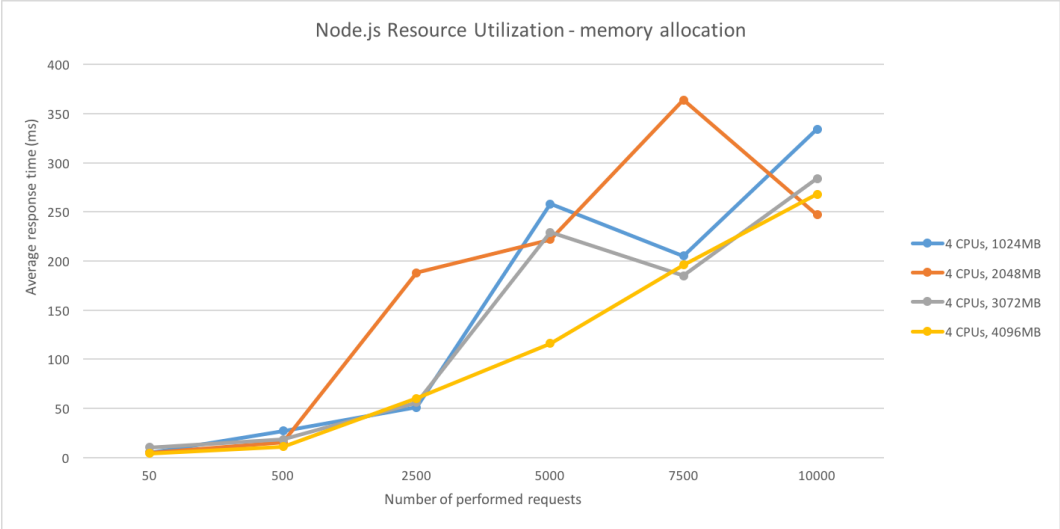


Figure 5.2: The average response time in milliseconds of the five API implementations for each of the six specified test cases

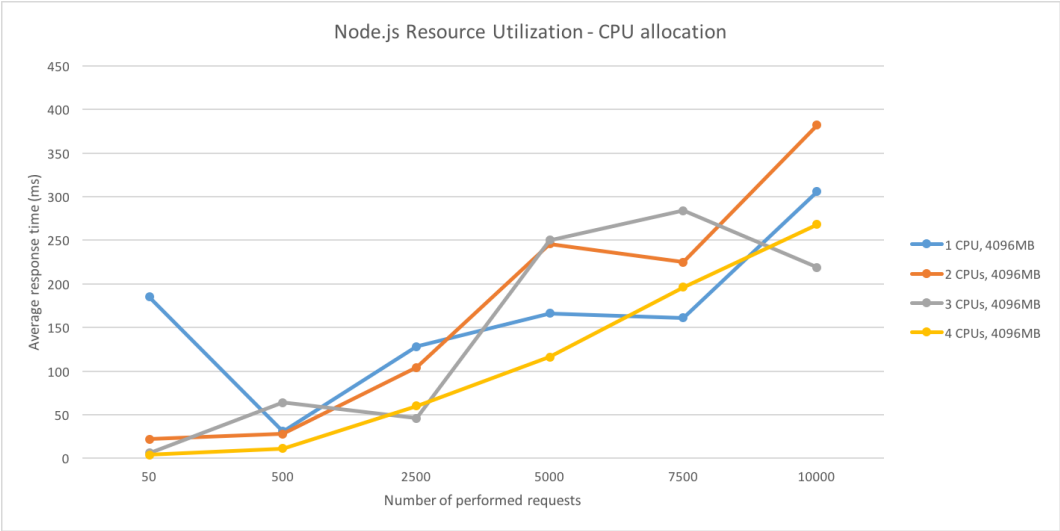
Figure 5.2 depicts a nearly linear average response time for the API implementation based on the Node.js technology, with the Swift-based implementation using the Perfect framework following closely behind. The PHP-based API displayed a slight linear increase in response time along with the increase in the number of concurrent requests, while both the Python and the Swift-Frank implementations depicted an almost exponential increase. From the sole perspective of time behavior, the Node.js technology proved to be the best suited for implementing server-side RESTful APIs, with the Swift-based implementation using Perfect as the underlying framework being a close second.

5.5.2 Resource utilization

The ability of each API implementation to effectively use its available resources in order to decrease its average response time was analyzed from two perspectives, namely from a varying amount of allocated memory, as well as a differing amount of available CPUs. In this case, each implementation was individually analyzed by performing two sets of tests, one with an optimal amount of memory allocated, namely 4096MB, and a varying number of CPUs, the other with an optimal number of four allocated CPUs and a varying amount of allocated memory. Each configuration required restarting and changing the virtual machine hardware specifications of the Docker host containing the server components. As with the previous time behavior analysis, each test was performed five times with differing numbers of concurrent API requests. The results for each API implementation were gathered and displayed in the following graphical representations:



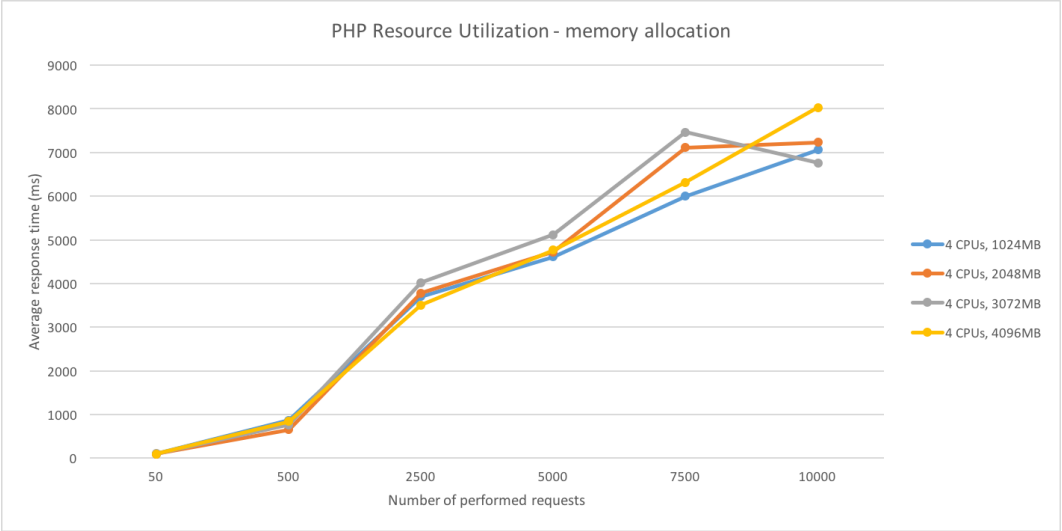
(a) Average response time of the Node.js-based server component having four CPUs allocated and a varying amount of memory



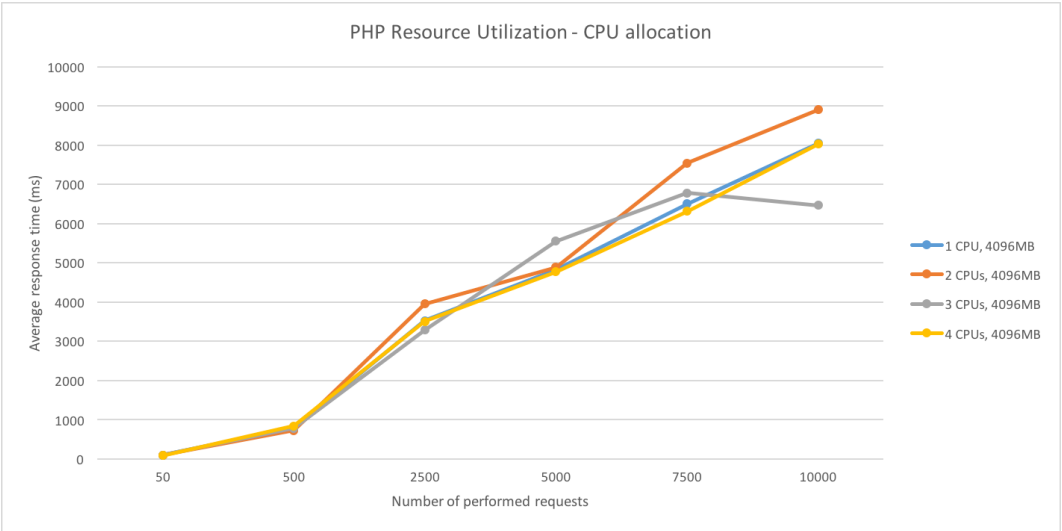
(b) Average response time of the Node.js-based server component having 4096MB allocated and a varying number of CPUs

Figure 5.3: Resource utilization of the Node.js-based API implementation

Figure 5.3 displays the behavior of the Node.js-based server component when confronted with a varying amount of allocated memory and number of CPUs, respectively. In both cases a slight overall improvement can be noted. At the same time, it can be observed that using the optimal configuration of 4096MB of allocated memory as well as four CPUs leads to a smoother increase in average response times, as opposed to all other cases.



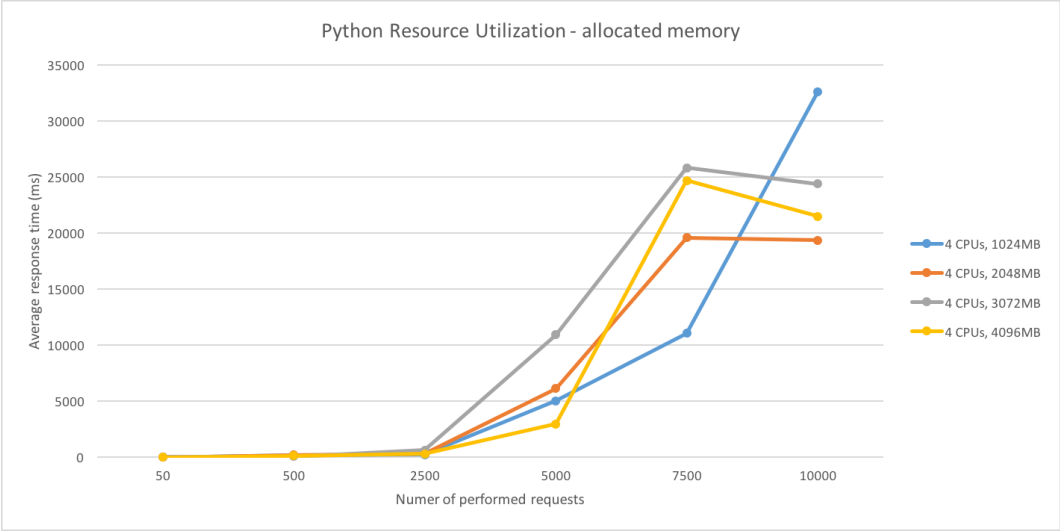
(a) Average response time of the PHP-based server component having four CPUs allocated and a varying amount of memory



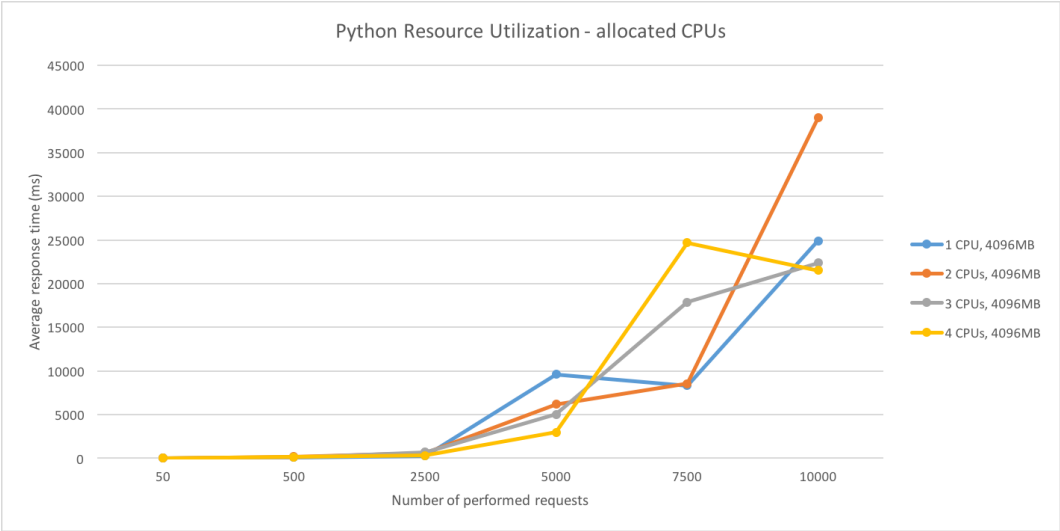
(b) Average response time of the PHP-based server component having 4096MB allocated and a varying number of CPUs

Figure 5.4: Resource utilization of the PHP-based API implementation

The above Figure 5.4 depicts the behavior of the PHP-based implementation when confronted with the same scenario. The two graphs show no significant change in average response time of the server component when more resources were allocated.



(a) Average response time of the Python-based server component having four CPUs allocated and a varying amount of memory

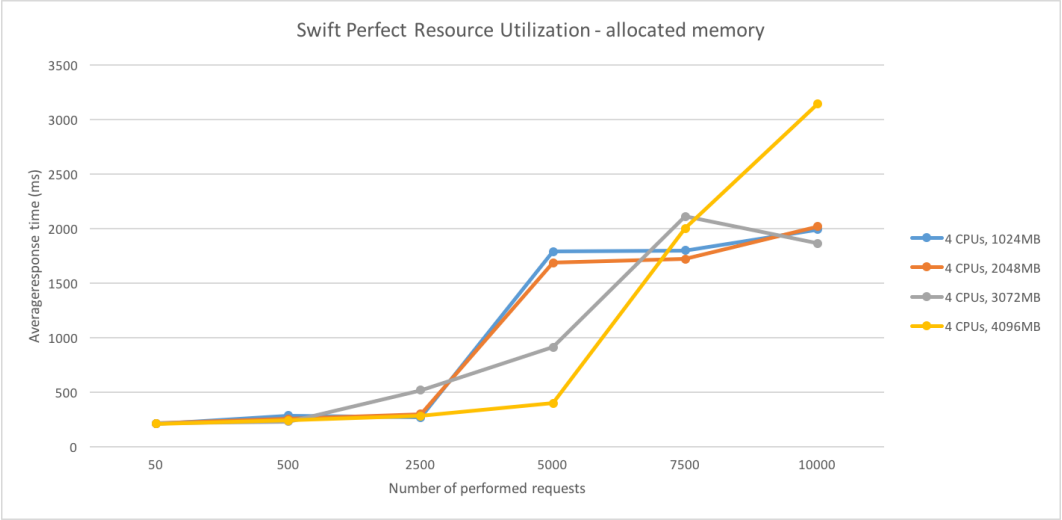


(b) Average response time of the Python-based server component having 4096MB allocated and a varying number of CPUs

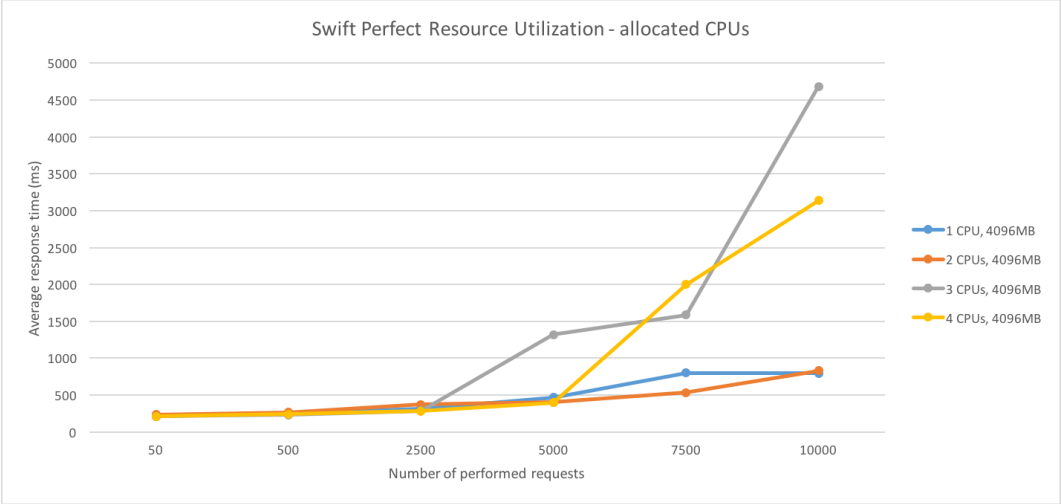
Figure 5.5: Resource utilization of the Python-based API implementation

As opposed to the previous two cases, the Python implementation registered an actual increase in average response times when being allocated more hardware resources, as can be seen in Figure 5.5. It is worth noting, however, that an increase in both memory and CPU number led to an increase in performance up until the 1000 concurrent requests[65].

[65] Due to the fact that each test was performed five times, the total number of requests performed is five times the number specified in each test configuration, corresponding to the values represented on the X-axes of all graphs.



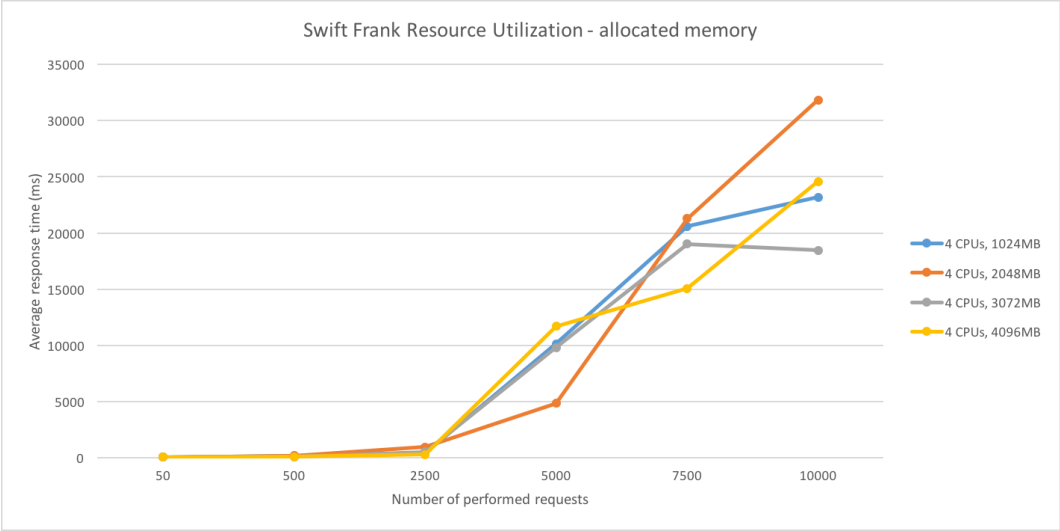
(a) Average response time of the Swift-based server component with Perfect as the underlying framework having four CPUs allocated and a varying amount of memory



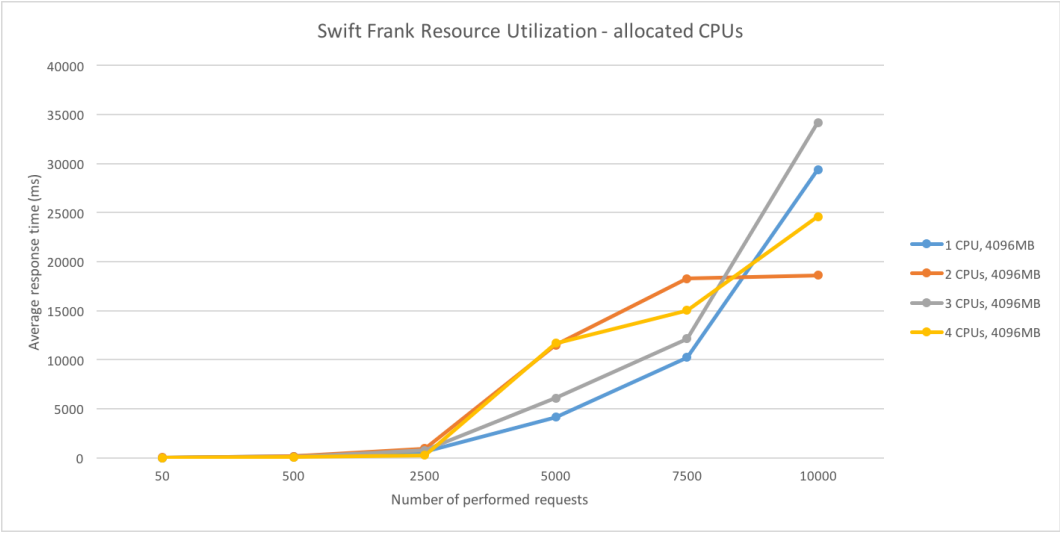
(b) Average response time of the Swift-based server component with Perfect as the underlying framework having 4096MB allocated and a varying number of CPUs

Figure 5.6: Resource utilization of the Swift-based API implementation having Perfect as its underlying framework

Similarly, in the case of the Swift implementation having the Perfect framework as its underlying technology, increasing the number of CPUs led to a slower average response time, as can be seen in Figure 5.6. The implementation performed better when it had a maximum of two CPUs allocated. On the other hand, increasing the allocated memory led to a significant decrease in average response time up until a number of 1000 concurrent requests.



(a) Average response time of the Swift-based server component with Frank as the underlying framework having four CPUs allocated and a varying amount of memory



(b) Average response time of the Swift-based server component with Frank as the underlying framework having 4096MB allocated and a varying number of CPUs

Figure 5.7: Resource utilization of the Swift-based API implementation having Frank as its underlying framework

Lastly, Figure 5.7 shows no significant change in average response time for the Swift-based implementation having Frank as its underlying framework, regardless of an increase in allocated memory or number of available CPUs.
Based on the average response times registered and represented for each of the five implementations having a varying CPU number and allocated memory, it can be concluded that the Node.js-based application utilizes its resources in the most advantageous way and is capable of increasing its performance when receiving an increase in hardware capabilities. On the other hand, all other technologies displayed no significant change in their average response times in order to be considered as

taking advantage of the available hardware resources, with the noteworthy exception of the Perfect-based Swift and the Python implementations, which registered a slight increase until reaching a number of 1000 concurrent requests.

5.6 Reliability

The reliability of a software system is defined by the ISO/IEC 25010 norm as its capability to maintain a certain performance when used under predetermined conditions. It consists of the following four sub-characteristics:

1. Maturity
   The maturity of a software system is defined by its capability to avoid failure as a result of faults in the software system itself. For the purposes of the current thesis, it is assumed that all five implementations of the API are correct and cannot experience software-related failure. The maturity was therefore not taken into consideration in the evaluation of the five API implementations.

2. Fault Tolerance
   This sub-characteristic concerns the frequency of failure of a given software system. In order to compare the fault tolerance of the five different API implementations, the error rate values stored in the Summary Report object of each JMeter test case were used. Further tests were conducted in order to analyze to what extent the fault tolerance of each implementation could be affected by changes in available hardware resources, similarly to the previously described resource utilization tests.

3. Recoverability
   This aspect is represented by the capability of a software system to be brought to full operation following a failure. As all five implementations are based on the same Docker container software and can be restarted in the same manner, this point was not taken into consideration when evaluating the reliability of the five server components.

4. Reliability Compliance
   Similarly to the previous sections, the compliance to external standards and conventions was not taken into consideration due to the fact that the five applications were developed solely for research purposes of their underlying technologies.

5.6.1 General fault tolerance

In order to compare the five applications based on their fault tolerance, tests were performed using the JMeter tool in order to measure the average error rate when

performing a varying number of concurrent requests. An optimal hardware configuration was chosen, namely 4096MB of allocated memory and four allocated CPUs, which were specified in the hardware configuration of the Docker virtual machine hosting the five server applications. At the same time, each server component was tested individually by shutting down every other Docker container for the duration of its tests. A total of six different tests were performed for each implementation, with the number of concurrent threads taking the values of 10, 100, 500, 1000, 1500 and 2000. The tests were performed five times and the resulting average error rates, calculated in percentages, were collected and displayed in the following graphical representation:

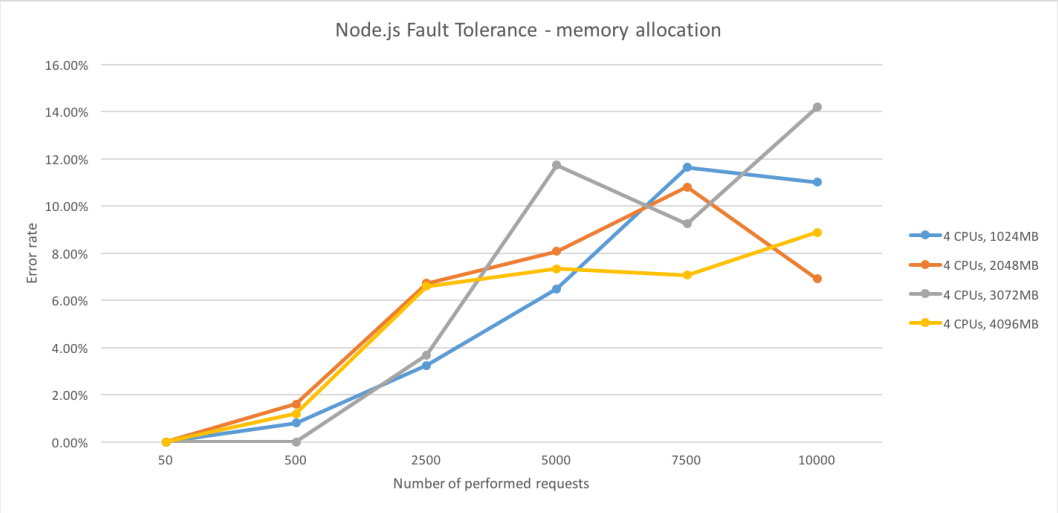
Figure 5.8: The error rate in percentage form displayed by each of the five API implementations for each of the six specified test cases

As can be seen in Figure 5.8, the Node.js-based implementation had the lowest average error rate, registering a maximum error rate of 8.87% when processing 2000 concurrent requests. It is also worth noting that, following a slight increase when confronted with more than 500 concurrent requests, the evolution of the error rate for the Node.js implementation follows an almost constant pattern. Conversely, the PHP, Python and Swift-Frank implementations displayed a higher error rate ranging between 28.05% and 41.98% when processing the maximum number of 2000 concurrent requests. The graph also shows a linear tendency of error rate growth for these three implementations. The outlier of the current analysis is the Swift-based application having the Perfect framework as its underlying technology. It registered the lowest error rate of 4.00% when processing 500 concurrent requests and an increase to 100% when receiving more than 1500 simultaneous requests. An error rate of 100% should be interpreted in the context of the current thesis as the inability of a software system to continue performing its task. While all other four applications continued to perform under the aforementioned conditions, the Swift-based applica-

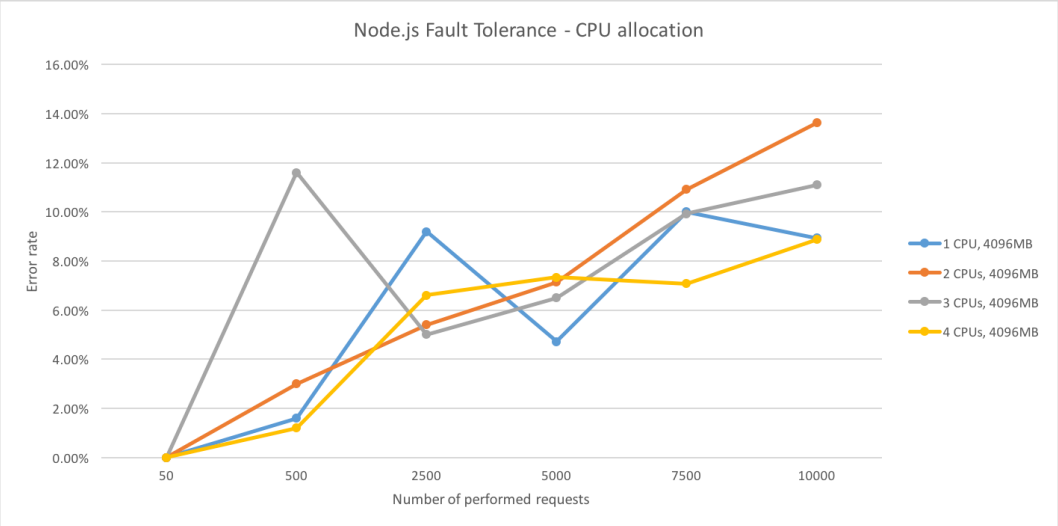
tion having Perfect as its underlying technology was terminated, therefore causing its corresponding Docker container to shut down and become unresponsive to any requests on the predefined port.

5.6.2 Influence of hardware resources

While the previous analysis provided an insight into the fault tolerance of the five API implementations, it was decided to also investigate whether different hardware configurations had any effect on the evolution of the error rates for each individual application.



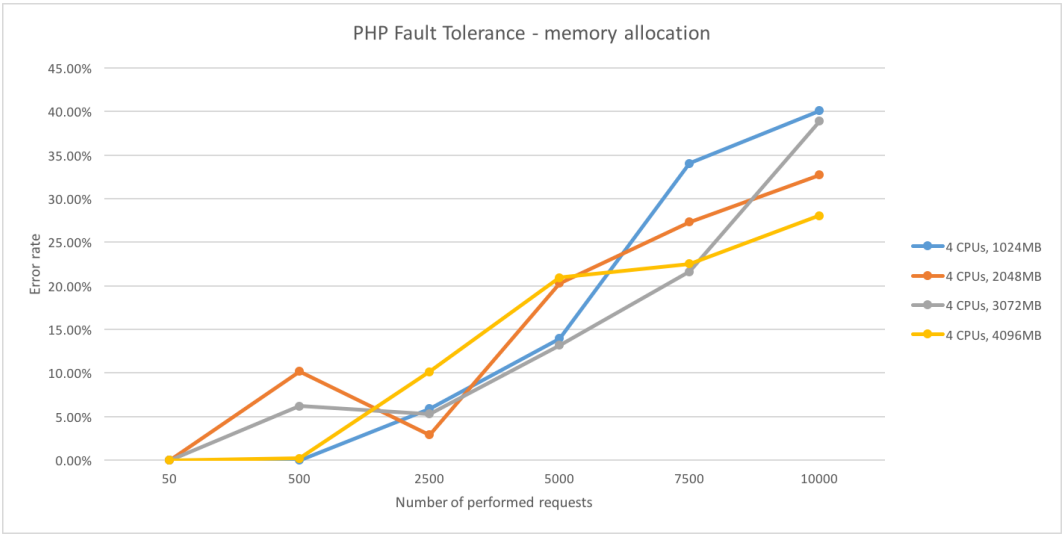
(a) Average error rate of the Node.js-based server component having four CPUs allocated and a varying amount of memory



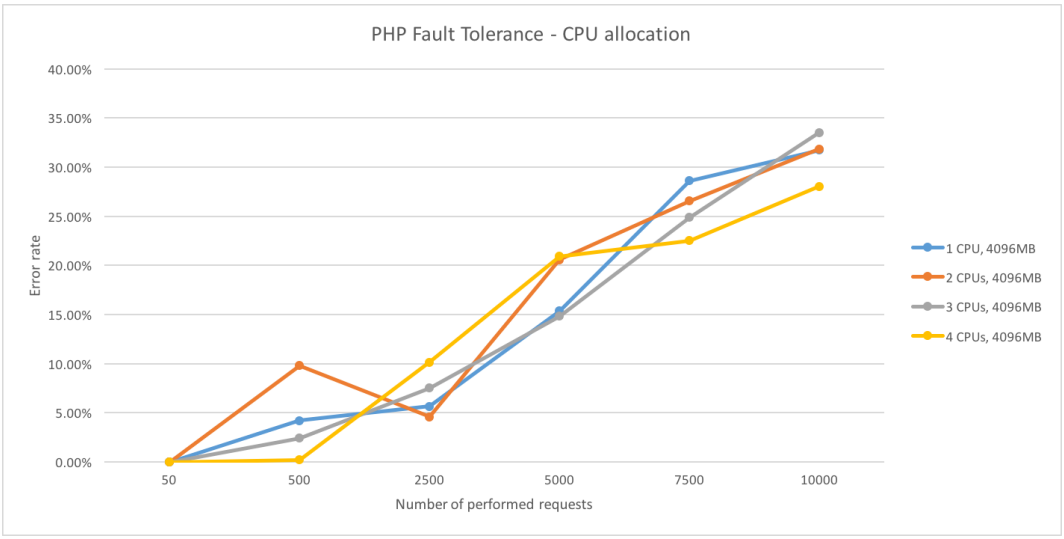
(b) Average error rate of the Node.js-based server component having 4096MB allocated and a varying number of CPUs

Figure 5.9: Evolution of the average error rate for the Node.js-based server component implementation

In the case of the Node.js implementation, increasing the amount of allocated memory, as well as increasing the number of available CPUs, led only to a minor improvement in the evolution of the error rate.



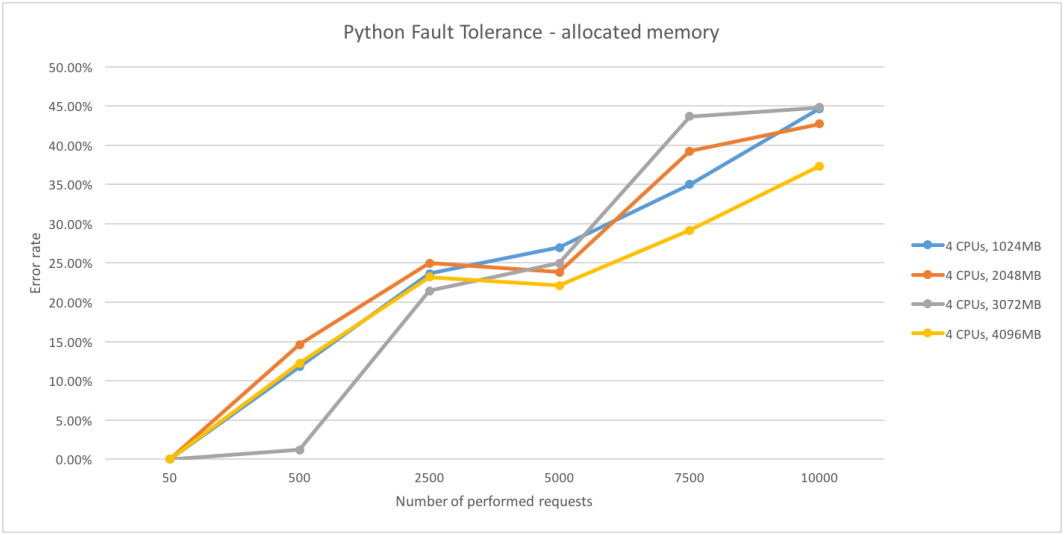
(a) Average error rate of the PHP-based server component having four CPUs allocated and a varying amount of memory



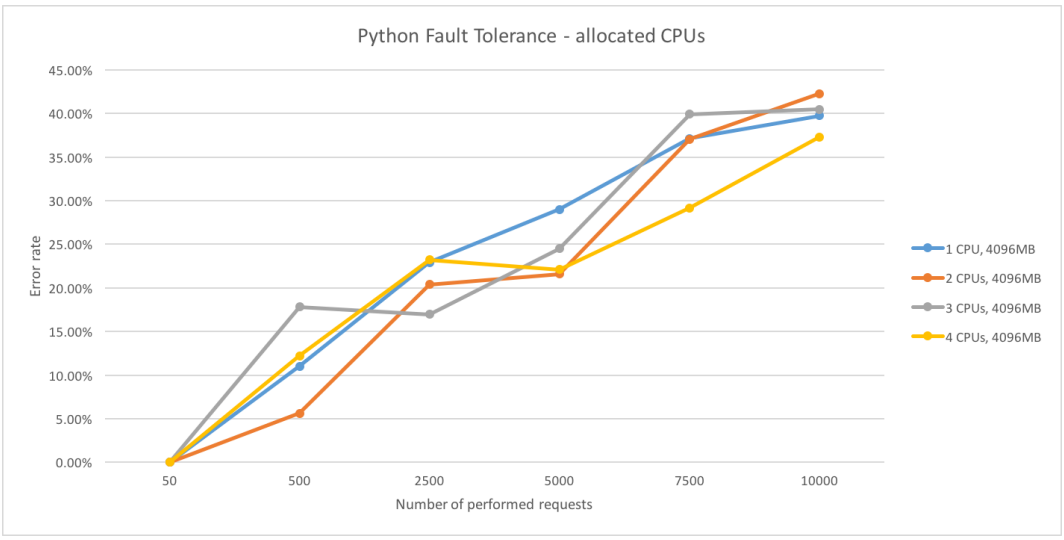
(b) Average error rate of the PHP-based server component having 4096MB allocated and a varying number of CPUs

Figure 5.10: Evolution of the average error rate for the PHP-based server component implementation

An increase in the amount of allocated memory and of number of available CPUs also had no significant effect on modifying the error rate of the PHP-based implementation of the server component.



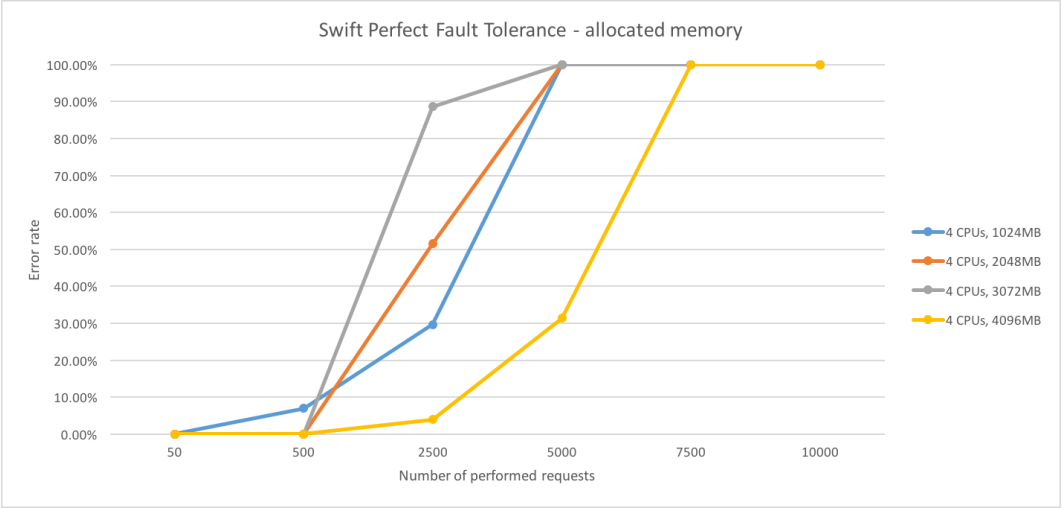
(a) Average error rate of the Python-based server component having four CPUs allocated and a varying amount of memory



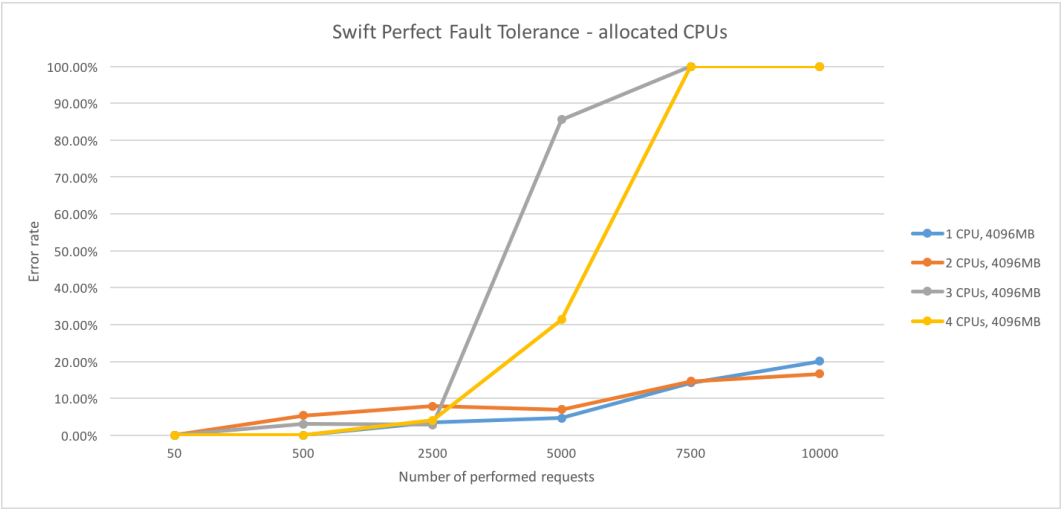
(b) Average error rate of the Python-based server component having 4096MB allocated and a varying number of CPUs

Figure 5.11: Evolution of the average error rate for the Python-based server component implementation

Similarly to the previous cases, an increase in the amount of memory and number of CPUs allocated to the virtual machine hosting the Python-based server component had no remarkable impact on the evolution of its error rate.



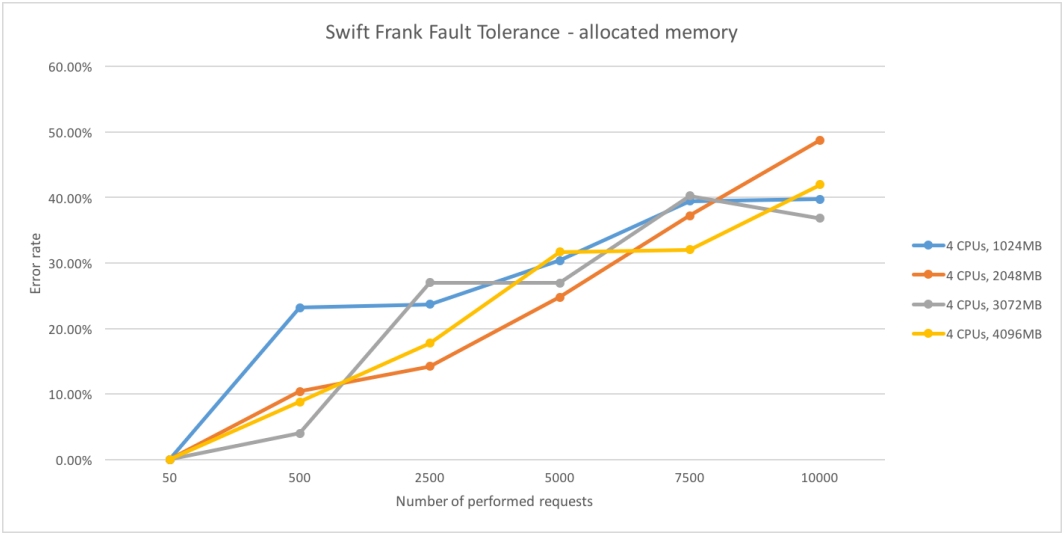
(a) Average error rate of the Swift-based server component having Perfect as its underlying framework, four CPUs allocated and a varying amount of memory



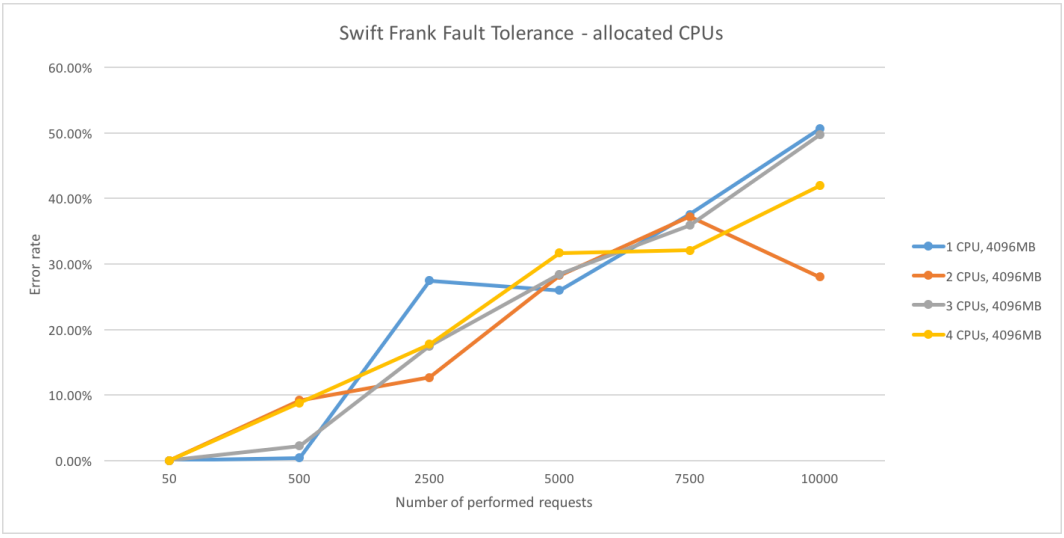
(b) Average error rate of the Swift-based server component having Perfect as its underlying framework, 4096MB allocated and a varying number of CPUs

Figure 5.12: Evolution of the average error rate for the Swift-based server component implementation having Perfect as its underlying framework

In the case of the Swift-based implementation having Perfect as its underlying technology, an increase of allocated memory lead to a significant reduction in error rate. While the implementations having up to 3072MB of allocated memory failed when confronted with 1000 concurrent requests, the implementation having 4096MB at its disposal delayed its failure until having to process 1500 simultaneous requests. On the other hand, allocating more than two CPUs to the virtual machine hosting the application led to a drastic increase in error rate for the Swift-based application. Allocating one or two CPUs, however, maintained the evolution of the error rate to a slight linear pattern.



(a) Average error rate of the Swift-based server component having Frank as its underlying framework, four CPUs allocated and a varying amount of memory



(b) Average error rate of the Swift-based server component having Frank as its underlying technology, 4096MB allocated and a varying number of CPUs

Figure 5.13: Evolution of the average error rate for the Swift-based server component implementation having Frank as its underlying framework

The Swift-based implementation having Frank as its underlying framework, on the other hand, showed no significant change in error tolerance when being allocated more hardware resources, whether memory or number of CPUs.
Based on the previously described graphics and the results that they depict, it can be concluded that the Node.js, PHP, Python and Swift-Frank implementations have no noticeable change in error rate when allocated an increased amount of hardware resources. On the other hand, the Swift-based implementation using the Perfect library as its underlying technology displays a positive response when being allocated more memory and increases its error rate when being allocated more than two CPUs.

6 Discussion

Based on the previously stipulated observations and test results, the current chapter will seek to provide an analytic discussion regarding the appropriateness of using the Swift language in order to implement RESTful APIs, as well as server-side applications in general.

6.1 Methodology for implementation and testing

In order to assess how Swift-based services compared to more widely used technologies in the context of RESTful API implementation, the current thesis defined a methodology for the implementation and testing of five individual server-side API components based on the JavaScript, PHP, Python and Swift languages respectively. The methodology was specified in order to ensure that the resulting five applications differed as little as possible from every aspect other than their underlying technologies. Any subsequently performed tests would therefore be a direct reflection of the technologies' appropriateness to implement RESTful APIs and not of any additional perturbing factor.
The resulting five applications were also tested solely from the perspective of previously defined analysis points, namely functionality, efficiency, reliability, disk usage and ease of implementation, all of which were chosen due to their relevance regarding the underlying technologies.

6.2 Formal evaluation

All five applications correctly returned the expected response when confronted with the predefined HTTP GET request, thus meeting their functionality requirement.
In terms of reliability, however, both Swift-based implementations displayed a relatively high error rate compared to the other technologies, with the Perfect-based application repeatedly shutting down upon receiving too many concurrent requests. Increasing the hardware resources proved to be slightly effective only for the Perfect-based server component, but did not sufficiently improve the general fault tolerance of the application in order to bring it on the same level as the Node.js-based implementation.
From the perspective of efficiency, the Frank-based server component proved to have a relatively low response time, but it did not reach a competitive level with the more

efficient Node.js-based server. On the other hand, it proved to be much more efficient than its Perfect-based counterpart and surpassed the performance of both the PHP and Python-based implementations.
Altering the allocated hardware resources also did not have a significant impact upon either the efficiency or the reliability of the two Swift-based implementations. Of the five server components, only the one based on Node.js displayed a tendency to improve its response times when being allocated with more memory and with a higher number of CPUs.
The two Swift-based components also required almost twice as much disk space as all other implementations, due to the high number of dependencies and the size of the Swift package itself.
From the perspective of ease of implementation, the Perfect-based application proved to be the most difficult to implement, while the Frank-based implementation required as little effort as all of its remaining counterparts.
It is worth noting how the underlying framework of the two Swift-based implementations influenced their behavior and corresponding test results. While the Perfect-based library proved to be more difficult to use and displayed an unacceptable error rate compared to its counterparts, the Frank-based implementation at times surpassed the more mature PHP and Python technologies in terms of efficiency, as well as almost matched them in terms of reliability. It can be speculated that this discrepancy is either determined by the underlying framework and its implementation, or by the fact that the Frank library uses a more recent version of the Swift language, namely 3.0, while the Perfect library continues to be based upon the more outdated 2.2 version.
Due to its unreliability, the Perfect-based application could not be deemed appropriate for the implementation of RESTful APIs. However, the Frank-based Swift implementation provided both appropriate response times and error rates in order to be deemed as a fitting technology to be used in the implementation of RESTful APIs and generally of server-side applications. While it does not represent the superior alternative, being slower, more error-prone and needing more disk space than Node.js, it does provide acceptable results when compared to the remaining technologies from the perspective of the previously mentioned analysis points. The Swift language can therefore be considered as appropriate to use in the implementation of RESTful APIs.

6.3 Drawbacks

Based on the formal evaluation of the five server components described in the previous chapters of the current paper, the Swift language has been declared as being appropriate for the development of server-side applications and more specifically for the implementation of RESTful APIs. On the other hand, its comparison with more

widely used technologies such as Node.js has also revealed that it is not yet ready for the implementation of server-side applications destined for production use. While proving to be faster than some of its counterparts in some instances, most notably surpassing Python implementations when using the Frank framework, both Swift-based API implementations were slow in comparison with the Node.js server-side API component. The same can be said for Swift's reliability and fault tolerance.
Since its release, the Swift language has registered a historic rise in developer interest and community participation, evolving and improving its syntax organically and at a high pace. This rapid evolution is not without its drawbacks, however. As mentioned in Chapter 4, one of the server-side applications developed with Swift used the Perfect library as its framework, which was based on Swift 2.2, while the other used Frank, which was compatible with the newer 3.0 version. However, both projects failed to compile when using the newest release of the language, which featured additional changes to the syntax that both frameworks did not yet include. Due to this rapid evolution in the Swift syntax, many external libraries fail to provide regular updates, an aspect that is not likely to change within the next year. It is therefore not advisable to implement a project destined for production with Swift yet, due to the currently volatile aspect of the language. A RESTful API is generally conceived to be a highly available service and as such should not be implemented with a language whose constant evolution and improvement would require regular refactoring of entire code bases, as well as increase the risk for potential service downtime when performing necessary updates.
A further drawback when using the Swift language for implementing RESTful APIs is its apparent lack of flexibility when confronted with additional hardware resources. A key factor in API development is the ability of the server component to react to a varying number of client requests and to take maximum advantage of the hardware resources it is provided with. While the Node.js server component showed an increase in both performance and fault tolerance when being allocated more memory and CPUs, both Swift implementations demonstrated minimal improvement when confronted with the same scenario. This behavior should not be attributed to the language itself, but rather to the two libraries used in order to implement the Swift-based server components. It can be assumed that, along with the evolution of the Swift language and its rise in popularity, more third-party libraries and components will be developed in order to meet the ever increasing demand for server-side Swift-based frameworks. At the time of this paper's release, however, no satisfactory libraries based on the Swift language existed.
Opting for Swift as the base technology for a server-side API component also poses a problem from the perspective of disk usage. The current tool ecosystem needed to run Swift-based applications in a Unix environment still requires more than one gigabyte of disk space, as opposed to the much more lightweight Python, Node.js and PHP toolkits. Using the Swift technology can be justified from this perspective

either when server-side disk use poses no problem, or if the technology were needed across multiple implementations of server-side applications, as opposed to being used in implementing only one instance of a RESTful API component.
Furthermore, the Swift tool ecosystem for Unix systems is currently only officially available for the Ubuntu operating system, versions 15.10 and 14.04. As briefly mentioned in Chapter 4, its use on a Debian operating system was attempted but failed due to missing dependencies. Therefore, depending on the operating system running on the server component used in order to implement a RESTful API, the use of the Swift technology may not even be a possibility in some instances.
Lastly, one of the main drawbacks when using Swift for implementing high-performing, reliable server-side applications is its demonstrated tendency to fail when confronted with overly stressful scenarios. Of the analyzed applications, the Swift-based component using the Perfect library as its base framework was the only process to crash when being presented with a high enough number of concurrent requests. And while using the Frank framework did have as an effect a more stable application without any crashes, the overall fault tolerance displayed by both instances was significantly lower than that of their counterparts. While it can be argued that the low fault tolerance can be attributed to the recent character of the language and to the immaturity of the libraries that were used to implement the Swift-based API components, it can be asserted that the technology itself, along with any third-party libraries based on it, is not yet suitable to implement reliable server-side applications destined for use in a production environment.

6.4 Advantages

Although using the Swift technology to implement production components may not be recommended, its potential use for implementing server-side applications used in a development environment should not be disregarded. The same rapid evolution of the technology that has led to multiple updates and which poses a risk to the stability of any Swift-based application in a production environment has also led to a rapid increase in the language's performance. While the first version of Swift was deemed to be slower even than its Objective-C predecessor, the newer versions have been declared to be reaching the performance of even C++ in some instances[66]. Being the only compiled programming language of the four technologies evaluated by the current paper and having the full support of Apple's engineering team and that of the open-source community, it can be assumed that Swift will continue to show a steady improvement rate in the foreseeable future. An ever faster and more evolved version of the language could therefore lead to it surpassing Node.js in terms of both performance and reliability.

[66] As per Anthony Schmieder. *Swift, C++ Performance*. Primate Labs. Dec. 3, 2014. URL: `https://www.primatelabs.com/blog/2014/12/swift-performance/` (visited on 07/08/2016)

Furthermore, the Swift Core Libraries[67] have yet to be released as part of the Swift tool ecosystem for the Ubuntu operating system. Their inclusion in the Swift Linux package will most likely lead to an even broader spectrum of potential server-side applications, given that they include the most widely used libraries and components for the development of iOS and OS X applications. At the time the current paper was written, the Swift Core Libraries were in the process of being ported by Apple's engineering team into the Swift Linux package and were set to be released in a future 3.0 version of the language. Their inclusion will give developers access to the Foundation framework, which provides a set of basic utility classes needed in the development of almost all applications. The package will also include the Grand Central Dispatch technology, which provides support for concurrent thread execution and multicore hardware. It can be assumed that, once GCD is made available for Swift in a Linux environment, third-party libraries such as the Perfect and the Frank frameworks will be able to improve their performance and fault tolerance when dealing with a high workload.

A further point to take into account when considering the use of Swift for implementing server-side Linux applications is the ease of implementation. The Swift language was designed to enable easy and fast programming[68] and supports multiple programming paradigms, such as object-oriented, protocol-oriented and functional programming. It is therefore flexible enough to enable most developers to easily switch from their standard programming language to implementing a Swift-based application. Furthermore, many RESTful APIs are designed to serve as the backend of a mobile application. The two most popular mobile application platforms are iOS and Android[69], the first of which uses Swift as its main development language. By implementing an API server component based on the same language as one of its clients, overall project costs can be reduced by eliminating the developer's need to either learn or get reacquainted with any additional server-side language.

Lastly, while the Swift language is still in its infancy and is constantly evolving, its unprecedented rise in popularity has prompted its consideration as a first class programming language not only for the implementation of iOS and OS X applications, but also for that of Android[70] and Windows mobile applications. Such a decision would have a dramatic impact on the market of mobile application developers, promoting the use of Swift as a cross-platform programming language for the implementation of both mobile applications and their corresponding server-side com-

[67] The Swift Core Libraries are briefly described on the official page of the Swift language under: https://swift.org/core-libraries/#foundation (visited on 30/06/2016)

[68] As stated by Apple.Inc in Apple Inc. *The Swift Programming Language (Swift 2.1)*. Cupertino, CA 95014: Apple Inc., 2014

[69] As of August 2015, Android and iOS had a combined market share of 96.7%, as documented under: http://www.idc.com/prodserv/smartphone-os-market-share.jsp (visited on 30/06/2016)

[70] See Nate Swanner. *Google is said to be considering Swift as a 'first class' language for Android.* Apr. 7, 2016. URL: `http://thenextweb.com/dd/2016/04/07/google-facebook-uber-swift/\#gref` (visited on 06/30/2016)

ponents.
Therefore, while Swift may not be ready for developing fully matured server components destined for use in production environments, its consideration as a viable alternative as a development language for server-side applications is strongly recommended due to its overwhelming popularity and its potential for becoming a standard tool in the developer community.

7 Conclusion

The present thesis has sought to analyze the appropriateness of using the Swift language in order to implement RESTful API components in a Linux environment, as well as to implement server-side applications in general. In order to provide an assessment of the technology's suitability for such a task, a comparison between it and more widely used technologies was performed. For this purpose, a methodology for the implementation and testing of five individual API components based on Node.js, PHP, Python and Swift was defined and followed, with the results of the performed tests being summarized and discussed in Chapter 5.

Based on the implementation process of the server-side components using the aforementioned technologies, as well as on the results gathered in their corresponding evaluation phase, the Swift language was deemed appropriate for the implementation of RESTful APIs. However, its use in production environments is discouraged due to its instability and low fault tolerance compared to the more mature Node.js alternative. Being a relatively new language and having displayed a steep evolution rate, Swift is not the most suitable alternative for the implementation of stable and reliable server-side applications. However, the unparalleled interest and contribution from the developer community, as well as the language's potential for surpassing its current competitors in the foreseeable future make Swift an ideal candidate for the implementation of both RESTful API components and more general server-side applications in development environments. Swift is therefore not yet ready for production use, but should be considered as a viable alternative for the implementation of future server-side applications.

List of Figures

Acronyms

API	Application Program Interface 1
CPU	Central Processing Unit 7
FTP	File Transfer Protocol 4
HTTP	Hypertext Transfer Protocol 4
JVM	Java Virtual Machine 40
PHP	PHP: Hypertext Preprocessor 7
REST	Representational State Transfer 1
SMTP	Simple Mail Transfer Protocol 4

Bibliography

[1] *Introduction to Swift*. Apple Inc. June 2, 2014. URL: https://developer.apple.com/videos/wwdc2014/ (visited on 03/16/2016).

[2] *TIOBE Index for Swift*. Tiobe. Mar. 16, 2016. URL: http://www.tiobe.com/tiobe_index?page=Swift (visited on 03/16/2016).

[3] *TIOBE Index for Objective-C*. Tiobe. Mar. 16, 2016. URL: http://www.tiobe.com/tiobe_index?page=Objective-C (visited on 03/16/2016).

[4] *Apple Releases Swift as Open Source*. Apple Inc. Dec. 3, 2015. URL: http://www.apple.com/pr/library/2015/12/03Apple-Releases-Swift-as-Open-Source.html (visited on 03/16/2016).

[5] *Client/Server Architecture*. Oracle Corporation. June 16, 2016. URL: https://docs.oracle.com/cd/A57673_01/DOC/server/doc/SCN73/ch20.htm (visited on 06/16/2016).

[6] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. University of California, Irvine, 2000.

[7] *Hypertext Transfer Protocol – HTTP/1.1*. Network Working Group. June 1, 1999. URL: https://tools.ietf.org/html/rfc2616 (visited on 06/16/2016).

[8] *Node.js v5.11.1 Documentation*. Node.js Foundation. June 16, 2016. URL: https://nodejs.org/dist/latest-v5.x/docs/api/ (visited on 06/16/2016).

[9] *PM2 - Advanced Node.js process manager*. Keymetrics. June 30, 2016. URL: http://pm2.keymetrics.io (visited on 06/30/2016).

[10] *Express - Node.js web application framework*. Node.js Foundation. June 30, 2016. URL: http://expressjs.com (visited on 06/30/2016).

[11] *Socket.IO*. June 30, 2016. URL: http://socket.io (visited on 06/30/2016).

[12] *PHP Manual*. The PHP Group. June 15, 2016. URL: http://php.net/manual/en/ (visited on 06/16/2016).

[13] *Python 3.5.1 documentation*. Python Software Foundation. June 10, 2016. URL: https://docs.python.org/3/ (visited on 06/16/2016).

[14] Sudhi Seshachala. *Docker vs VMs*. Nov. 24, 2014. URL: http://devops.com/2014/11/24/docker-vs-vms/ (visited on 06/16/2016).

[15] *Linux Containers*. Canonical Ltd. June 30, 2016. URL: https://linuxcontainers.org (visited on 06/30/2016).

[16] Karl Seguin. *Node.js, Require and Exports*. Feb. 3, 2012. URL: http://openmymind.net/2012/2/3/Node-Require-and-Exports/ (visited on 06/30/2016).

[17] *curl and libcurl*. June 30, 2016. URL: https://curl.haxx.se (visited on 06/30/2016).

[18] *Composer*. June 30, 2016. URL: https://getcomposer.org (visited on 06/30/2016).

[19] *Slim Framework - Slim Framework*. June 30, 2016. URL: http://www.slimframework.com (visited on 06/30/2016).

[20] *pip 8.1.2 : Python Package Index*. June 30, 2016. URL: https://pypi.python.org/pypi/pip (visited on 06/30/2016).

[21] *Flask (A Python Microframework)*. June 30, 2016. URL: http://flask.pocoo.org (visited on 06/30/2016).

[22] *PEP 318 – Decorators for Functions and Methods*. June 5, 2016. URL: https://www.python.org/dev/peps/pep-0318/ (visited on 06/30/2016).

[23] *Perfect: Server-Side Swift*. June 5, 2016. URL: https://github.com/PerfectlySoft/Perfect (visited on 06/30/2016).

[24] *Frank*. June 5, 2016. URL: https://github.com/nestproject/Frank (visited on 06/30/2016).

[25] *Apache JMeter*. June 5, 2016. URL: http://jmeter.apache.org (visited on 06/30/2016).

[26] *ISO/IEC 25010:2011(en) Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*. 2011.

[27] Anthony Schmieder. *Swift, C++ Performance*. Primate Labs. Dec. 3, 2014. URL: https://www.primatelabs.com/blog/2014/12/swift-performance/ (visited on 07/08/2016).

[28] Apple Inc. *The Swift Programming Language (Swift 2.1)*. Cupertino, CA 95014: Apple Inc., 2014.

[29] Nate Swanner. *Google is said to be considering Swift as a 'first class' language for Android*. Apr. 7, 2016. URL: http://thenextweb.com/dd/2016/04/07/google-facebook-uber-swift/\#gref (visited on 06/30/2016).

Declaration

I hereby declare that the present paper and the work reported herein was composed by and originated entirely from me without any help. All sources used from published or unpublished work of others are reported in the list of references. All parts of my work that are based on others' work are cited as such. The Bachelor's Thesis has not been submitted for any degree or other purposes, neither at the TH Köln – University of Applied Sciences nor at any other university or college.

Gummersbach, July 11th 2016

Teodora-Roxana Petrisor